Summer 9-1-2014

Supervised Machine Learning Under Test-Time Resource Constraints: A Trade-off Between Accuracy and Cost

Zhixiang Xu
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Kilian Q. Weinberger, Chair
Sanmay Das
Yasutaka Furukawa
Nan Lin
Robert Pless
Alice X. Zheng

Supervised Machine Learning Under Test-Time Resource Constraints:

A Trade-off Between Accuracy and Cost

by

Zhixiang (Eddie) Xu

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

August 2014
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to express my sincerest gratitude to my advisor Kilian Weinberger. His patience during my early Ph.D. years, his trust in me during my dark days, and his support and guidance during my entire course of study have always encouraged me and shaped my scientific thinking of machine learning. Without him I could not have written this thesis.

I would like to thank my committee Sanmay Das, Yasutaka Furukawa, Nan Lin, Robert Pless and Alice Zheng for their comments and suggestions. Lots of work in this thesis benefited from their insightful thoughts and generous contributions. I would also like to especially thank Robert Pless for his guidance during the very first semester of my Ph.D. study and Alice Zheng for providing me a great internship opportunity at Microsoft Research.

I would like to thank Olivier Chapelle and Fei Sha for collaborating throughout and providing me experiences and ideas about the field of machine learning. I would also like to thank my co-authors Minmin Chen, Jake Gardener, Dor Kedem, Gao Huang, and Matt Kusner for exchanging ideas, discussing papers and picking up my mistakes. Especially many thanks to Minmin for sharing her knowledge and thoughts and Matt for helping all the time. I would also like to thank my lab mates, Wenlin Chen, Yuzong Liu, Stephen Tyree, Wenlin Wang and Quan Zhou for taking time to help me debugging, installing packages, and preparing talks and posters. Especially thanks to Stephen Tyree for reviewing this thesis and developing tools that lots of my work is based on.

I would like to thank my parents, Yudi Xu and Shumin Li for their love and support throughout my life. I am also very grateful for my grandparents Enqing Li and Xiuran Ma. Without their careful parenting during my young age, I would not have gone so far.

Finally, I would like to thank my wife Wuxuan Xiang for sharing my ups and downs with love and support throughout the last few years.

Zhixiang (Eddie) Xu

Washington University in Saint Louis
August 2014

ABSTRACT OF THE DISSERTATION

Supervised Machine Learning Under Test-Time Resource Constraints:

A Trade-off Between Accuracy and Cost

by

Zhixiang (Eddie) Xu

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2014

Research Advisor: Professor Kilian Q. Weinberger, Chair

The past decade has witnessed how the field of machine learning has established itself as a necessary component in several multi-billion-dollar industries. The real-world industrial setting introduces an interesting new problem to machine learning research: computational resources must be budgeted and cost must be strictly accounted for *during test-time*. A typical problem is that if an application consumes x additional units of cost during test-time, but will improve accuracy by y percent, should the additional x resources be allocated? The core of this problem is a trade-off between accuracy and cost. In this thesis, we examine components of test-time cost, and develop different strategies to manage this trade-off.

We first investigate test-time cost and discover that it typically consists of two parts: feature extraction cost and classifier evaluation cost. The former reflects the computational efforts of transforming data instances to feature vectors, and could be highly variable when features are heterogeneous. The latter reflects the effort of evaluating a classifier, which could be substantial, in particular nonparametric algorithms. We then propose three strategies

to explicitly trade-off accuracy and the two components of test-time cost during classifier training.

To budget the feature extraction cost, we first introduce two algorithms: *GreedyMiser*[132] and *Anytime Representation Learning (AFR)*[135]. GreedyMiser employs a strategy that incorporates the extraction cost information during classifier training to explicitly minimize the test-time cost. AFR extends GreedyMiser to learn a cost-sensitive feature representation rather than a classifier, and turns traditional Support Vector Machines (SVM) [110] into test-time cost-sensitive anytime classifiers. GreedyMiser and AFR are evaluated on two real-world data sets from two different application domains, and both achieve record performance.

We then introduce *Cost Sensitive Tree of Classifiers (CSTC)*[134] and *Cost Sensitive Cascade of Classifiers (CSCC)*[137], which share a common strategy that trades-off the accuracy and the *amortized* test-time cost. CSTC introduces a tree structure and directs test inputs along different tree traversal paths, each is optimized for a specific sub-partition of the input space, extracting different, specialized subsets of features. CSCC extends CSTC and builds a linear cascade, instead of a tree, to cope with class-imbalanced binary classification tasks. Since both CSTC and CSCC extract different features for different inputs, the amortized test-time cost is greatly reduced while maintaining high accuracy. Both approaches out-perform the current state-of-the-art on real-world data sets.

To trade-off accuracy and high classifier evaluation cost of nonparametric classifiers, we propose a model compression strategy and develop *Compressed Vector Machines (CVM)*. CVM focuses on the nonparametric kernel Support Vector Machines (SVM), whose test-time evaluation cost is typically substantial when learned from large training sets. CVM is a post-processing algorithm which compresses the learned SVM model by reducing and

optimizing support vectors. On several benchmark data sets, CVM maintains high test accuracy while reducing the test-time evaluation cost by several orders of magnitude.

Chapter 1

Introduction

Machine learning, a relatively new branch of artificial intelligence, studies systems that can learn from past experience. The past experience is commonly in a form of large amount of data, also referred to as training data. In a typical *supervised learning* scenario, training data contains pairs of instances in the form of features and outcomes, such as historical stock price and current stock price or clinical measurements and diabetes diagnostics. Using this training data, we build a prediction system (classifier) which can predict the outcomes from instance features, and we use the built prediction system to predict outcomes of previously unseen instances. It is referred to as *supervised learning* because outcomes are provided to guide the training process.

Consider an e-mail spam filtering application, where the goal is to build a classifier to predict if a new incoming e-mail is spam or not before delivering it to users' inboxes. To build such a classifier, a large amount of training data is collected. In this example, the training data include e-mails (instances) and their corresponding labels as spam or not spam (outcomes). E-mail instances are formulated as quantitative *feature vectors* readable by computers. Features may include words from the subject line and e-mail body, sending time, attachment types, attachment sizes in bytes, sender I.P., and spamming reputation. Figure 1.1 shows an example of the feature vector of one hypothetical e-mail instance.

A classifier is then learned to reproduce the outcome labels based on the instance features in the training data. The classifier determines which features to use and how to use them. A simple classifier might use the rule shown in Algorithm 1. This simple rule basically counts the number of occurrences of the keyword "viagra", and checks if the sender is in the user's

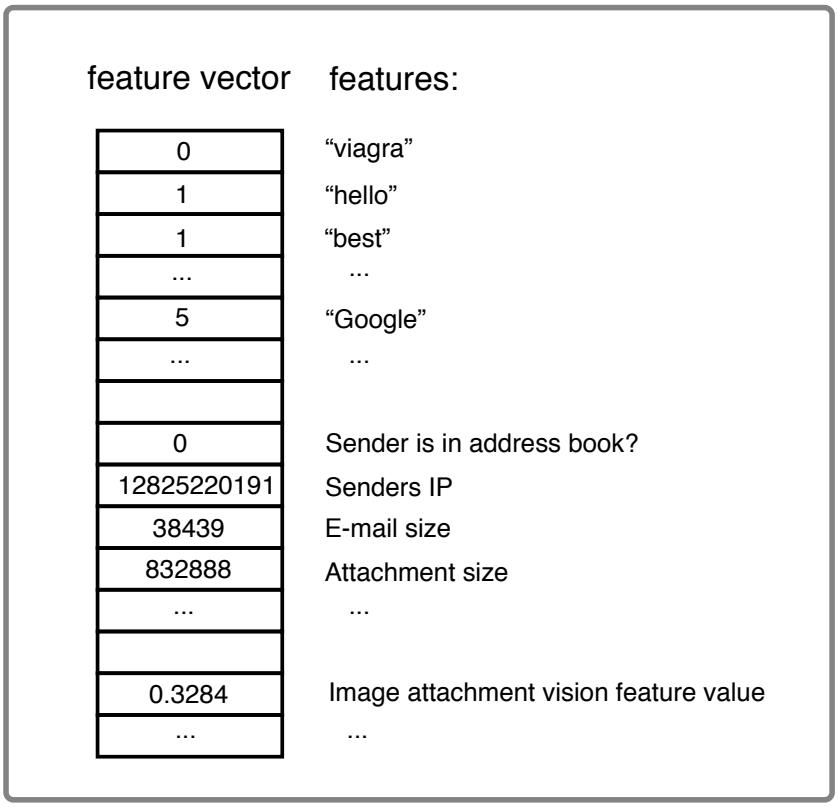

Figure 1.1: Features and the feature vector of one hypothetical e-mail instance.

address book. If the occurrence is greater than one *and* the sender is a stranger, the e-mail is classified as spam.

After training, the learned classifier is applied to classify previously unseen e-mails, a process called test-time evaluation or testing. To classify a new e-mail, one has to convert a data instance (a new e-mail in raw input) to a quantitative feature vector like described above, in the format required by the classifier. This stage is called *feature extraction*. Once features are extracted and concatenated into a feature vector, the learned classifier performs some computation to generate final predictions. This stage is called *classifier evaluation*. Another simple example of classifier evaluation which executes a classifier prediction rule is shown in Algorithm 2. The rule assigns a weight to each feature, where the weight is learned during training. Then the rule sums the weighted features and if the sum exceeds a threshold, it is classified as spam and otherwise as non-spam.

Algorithm 1 E-mail spam filtering rule 1

```
if ("viagra" ≥ 1) and (Sender is in address book == 0) then
  return SPAM
else
  return REGULAR
end if
```

Algorithm 2 E-mail spam filtering rule 2

```
if 0.9×"viagra" + 0.2×"hello" ≥ 3 then
  return SPAM
else
  return REGULAR
end if
```

Performing feature extraction and classifier evaluation during testing is not free and each stage described above incurs some certain cost. *Feature extraction cost* reflects the computational efforts of converting data instances to readable feature vectors. For example, counting the number of occurrences of keywords requires a full scan of the e-mail body, while extracting vision features from attached images requires running some vision algorithms. *Classifier evaluation cost* reflects the computation of generating predictions from feature vectors. For example, in Algorithm 2, the prediction rule requires multiply and sum operations, which consume CPU computation cost. These two costs combine to form test-time cost.

In the traditional machine learning setting, where data set sizes are small and classifiers are usually only executed once, test-time cost is low, and the sole goal is high classification accuracy. However, as machine learning enters into industry through applications such as web-search engines [142], product recommendation [40], and e-mail and web spam filtering [128], the setting becomes different. In all these applications, data set sizes are very large and classifiers are executed millions of times everyday. The test-time cost becomes an equally important concern as accuracy. Imagine a classifier that is executed 10 million times per day. We would like to introduce a new feature that improves the accuracy by 3%, but its extraction increases the running time by $1s$ per execution. 10 million executions would require the project manager to purchase 58 days of additional CPU time per day. Imagine another example where in order to classify a new e-mail, a classifier has to compare the feature vector of the new e-mail against that of all training e-mails (*e.g.* 10 million e-mails). Introducing additional 10 million training e-mails improves the accuracy of the classifier by

1%, but also significantly increases the test-time evaluation cost, as the classifier evaluation cost is linear in the number of training inputs. From these two large-scale real-world applications, it is clear that the real-world industrial setting introduces a new problem to machine learning research: computational resources must be budgeted and costs must be strictly accounted for *during test-time*. At its core, this problem is an inherent *trade-off* between accuracy and test-time cost.

In this thesis, we systematically investigate the test-time cost, quantify it, and propose four new approaches to explicitly control it under budget. To start, we first formally describe supervised learning and learning under test-time resource constraints. We then introduce some useful background in machine learning and give an overview of our four different approaches. In Chapter 2, we introduce two related algorithms that employ a strategy trading-off accuracy and test-time feature extraction cost. In Chapter 3 we describe another strategy, classification with trees and cascades, aiming to budget the amortized test-time cost. Chapter 4 targets the classifier evaluation cost and introduces an algorithm that explicitly controls it. Finally, Chapter 5 offers concluding remarks.

1.1 Learning and Learning Under Test-time Constraints

1.1.1 Supervised learning

Let $\mathbf{x}_i \in \mathcal{X}$ denote a training input in the form of a feature vector of dimension d, $\mathbf{x}_i \in \mathcal{R}^d$, with label $y_i \in \mathcal{Y}$. In supervised learning, training data are i.i.d. (independent and identical distributed) samples from a joint distribution $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ of instance/label pairs, *i.e.* $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. Labels $\mathcal{Y}$ can be binary, categorical or real numbered. E-mail spam filtering for example has binary labels, where each e-mail is either labeled as positive (regular) or negative (spam). Problems with binary, categorical and real number labels are referred to as binary classification, multi-class classification and regression problems, respectively.

Given the set of inputs and the corresponding labels, it is assumed that there is an underlying function f that maps the inputs to labels, $y_i = f(\mathbf{x}_i)$. The core of supervised learning is

to estimate this underlying function by learning an approximate hypothesis $H \in \mathbb{H}$ from a large amount of training inputs and their labels, and this hypothesis H is called *classifier*. Typical supervised learning methods include Logistic regression [53] and Support Vector Machines (SVM) [110, 29] for classification problems, and Neural networks [55] and Kernel regression [84] for regression problems.

1.1.2 Supervised learning under test-time resource constraints

When learning under test-time resource constraints, there are a test-time budget B and a test-time cost c. The test-time cost can be divided into feature extraction cost c_f and classifier evaluation cost c_e, corresponding to two stages during testing. A classifier's intrinsic structure (its prediction rule or algorithm) determines the evaluation cost, and thus the evaluation cost is a function of a specific classifier H, $c_e(H)$. We also assume that during testing, features are extracted *on-demand*, where features are only extracted from data instances when needed by the classifier. Therefore, a classifier H determines which features to extract, and the extraction cost is also a function of a specific classifier, $c_f(H)$.

This test-time cost and budget dramatically transform supervised learning. Instead of just learning a classifier H to maximize classification accuracy, one should also take test-time cost and budget into consideration during learning, making sure that the test-time cost of a classifier will be within budget constraints, $c_f(H) + c_e(H) \leq B$.

1.2 Types of Classifiers

Since a classifier's intrinsic structure dramatically affects the test-time cost, we review different types of classifiers. In general, a classifier H is learned by minimizing a loss function ℓ w.r.t. the classifier,

$$H = \min_{H} \ell(H). \tag{1.1}$$

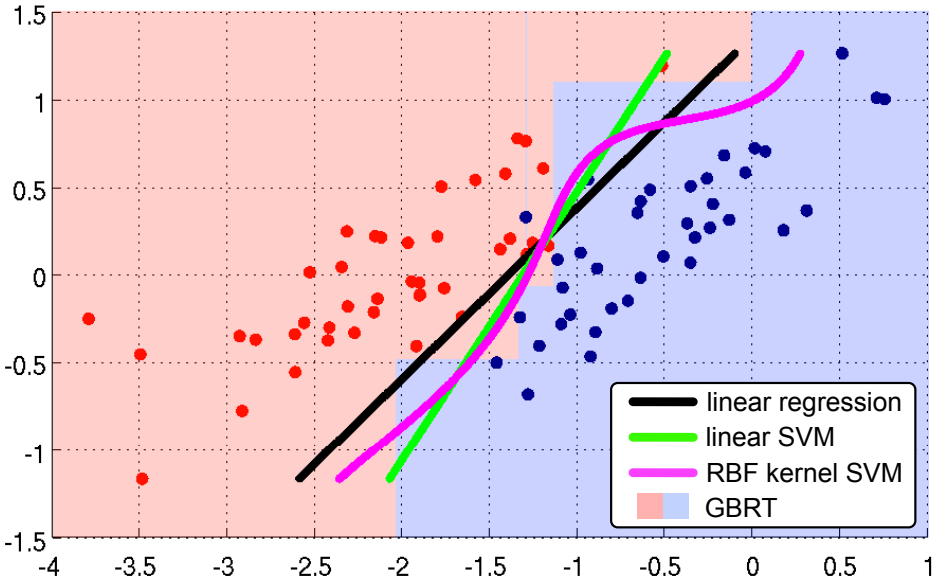

Figure 1.2: Decision boundaries of linear regression, linear SVM, kernel SVM and GBRT.

One example for ℓ is the squared-loss

$$\ell_{sq}(H) = \frac{1}{2n}\sum_{i=1}^{n}(H(\mathbf{x}_i) - y_i)^2, \tag{1.2}$$

but other losses, for example log-loss [53], are equally suitable.

1.2.1 Linear classifier

Linear classifiers have long been a popular classifier in statistics and machine learning, and still remain as an indispensable tool today. A linear classifier predicts an observed outcome y_i from a feature vector $\mathbf{x}_i \in \mathcal{R}^d$ using the model,

$$H(\mathbf{x}_i) = \mathbf{x}_i^\top \mathbf{w} + b, \tag{1.3}$$

where $H(\mathbf{x}_i)$ is the prediction of the observed outcome y_i, $\mathbf{w} \in \mathcal{R}^d$ is the weight vector assigning each feature a weight, and b is the bias. In the $(d+1)$-dimensional feature-prediction

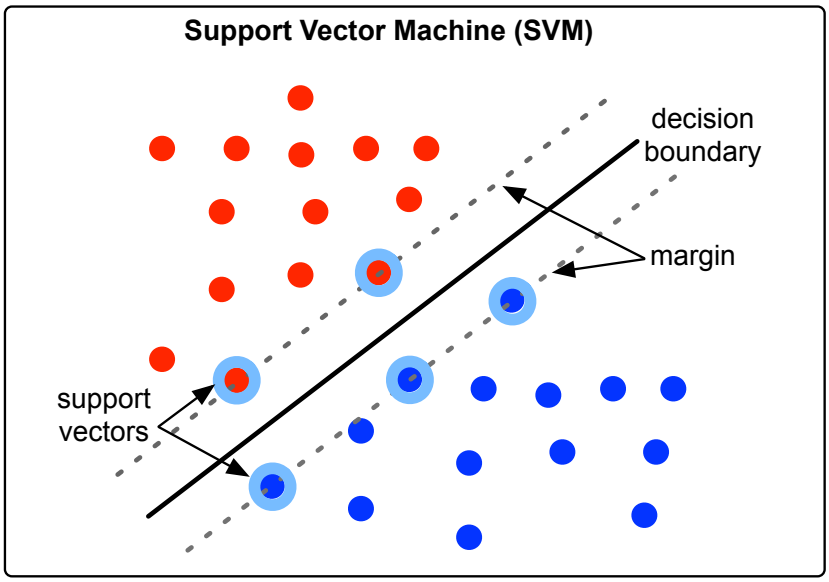

Figure 1.3: A schematic layout of Support Vector Machine (SVM). Blue and red dots are training instances in a two dimensional feature space, and two colors indicate two classes. The solid black line is the decision boundary learned from the training data. Dotted dark lines are margins. Dots in light blue circles are support vectors.

space, $\big(\mathbf{x}, H(\mathbf{x})\big)$ represents a hyperplane. This hyperplane is called a *decision boundary* for classification problems, as instances above the hyperplane are classified as positive and those below are classified negative. This decision boundary is parameterized by the weight vector $\mathbf{w}$, and it is learned by minimizing a squared-loss function

$$\min_{\mathbf{w}} \ell(w) = \sum_{i=1}^{n} (\mathbf{x}_i^\top \mathbf{w} + b - y_i)^2. \tag{1.4}$$

Figure 1.2 (black curve) shows a linear classifier and its decision boundary learned from a 2-dimensional Iris [2] data set. Commonly used linear classifiers include logistic regression and linear regression. In terms of the test-time cost, a linear classifier just needs to perform the inner product computation in (1.3), so the evaluation cost is very low.

1.2.2 Large margin classifier

To improve the generalization performance to previously unseen test data, Cortes and Vapnik [29] introduce Support Vector Machines (SVM). Compared to a regular linear model as described above, SVM enforces a large margin, maximizing the margin between the decision boundary and the closest training instances. Mathematically, the SVM decision boundary can be learned by solving an optimization problem:

$$\begin{aligned} &\min_{\mathbf{w},b} \|\mathbf{w}\| \\ &\text{s.t. } y_i(\mathbf{x}_i^\top \mathbf{w} + b) \geq 1, i = 1, \ldots, n, \end{aligned} \tag{1.5}$$

where the objective function maximizes the margin, and the constraints enforce that the decision boundary is at least one unit away from training instances. One key advantage of the SVM is that its decision boundary is completely defined by training instances on the margin, denoted *support vectors*. Since the number of support vectors is usually less than training instances, SVM is also referred to as *Sparse Vector Machine*. Figure 1.3 illustrates the margin, decision boundary and support vectors. Figure 1.2 (green curve) shows the decision boundary of an SVM on the Iris data set. To make predictions, an SVM uses the prediction rule:

$$H(\mathbf{x}_i) = \text{sign}(\mathbf{x}_i^\top \mathbf{w} + b). \tag{1.6}$$

The test-time evaluation cost is the inner product computation and is very low.

1.2.3 Kernel classifier

While the large margin enforcement provides better generalization on unseen test data, it is still restricted by its *linear* decision boundary, and is unable to handle linearly un-separable data. To overcome this, Guyon et al. [52] propose *kernel* SVM. Kernel SVM allows the algorithm to find the maximum-margin hyperplane in a transformed feature space ($\mathbf{x} \rightarrow \phi(\mathbf{x})$, where $\phi(\mathbf{x}) \in \mathcal{R}^D$, and $D \gg d$). The transformation enlarges the feature space and may be nonlinear. Therefore, while the resulting decision boundary is still a linear hyperplane in the high-dimensional feature space, it may be nonlinear in the original input space. To learn

such a hyperplane, kernel SVM optimizes the following problem:

$$\min_{\alpha_1,\ldots,\alpha_n} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{K}_{ij} - \sum_{i=1}^{n}\alpha_i, \tag{1.7}$$
$$\text{s.t. } \sum_{i=1}^{n}\alpha_i y_i = 0 \text{ and } \alpha_i \geq 0, i = 1,\ldots,n,$$

where $\boldsymbol{\alpha}$ are the Lagrange multipliers [6], and $\mathbf{K}$ is the kernel matrix whose entry $\mathbf{K}_{ij}$ is the inner product of the instances in the transformed space, $\mathbf{K}_{ij} = \phi(\mathbf{x}_i)^\top\phi(\mathbf{x}_j)$. Compared to other non-linear transformation [27, 107] the key advantage of this formulation is that one never needs to express $\phi(\mathbf{x})$ explicitly, instead using the kernel function $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ to implicitly transform the feature space. Note that the above optimization is equivalent to (1.5), only expressed in dual form [6] with the implicit feature transformation $\phi(\mathbf{x})$.

The kernel SVM prediction function is different from linear SVM,

$$H(\mathbf{x}_i) = \sum_{j=1}^{n}\alpha_j y_j \mathbf{K}_{ji} + b, \tag{1.8}$$

where $\mathbf{K}_{ji}$ is one kernel entry, which is the value of a kernel function of a test input $\mathbf{x}_i$ and one support vector, $\mathbf{K}_{ji} = k(\mathbf{x}_j, \mathbf{x}_i)$. Figure 1.2 (magenta curve) shows the non-linear decision boundary of a kernel SVM. Since kernel SVM evaluation involves computing kernel function of a test input and all its support vectors, its classifier evaluation cost is significantly higher than linear classifiers. Other popular kernel classifiers include kernel regression [66] and Gaussian processes [98].

1.2.4 Tree-based classifier

Another set of classifiers are tree-based classifiers. They all use decision trees to partition the feature space into a set of rectangles, on which they train simple and weak classifiers. These weak classifiers are also referred to as *weak learners*. While conceptually simple, tree-based classifiers are powerful and produce non-linear decision boundaries. Popular tree-based classifiers include random forest [8] and gradient boosted regression trees (GBRT)[44].

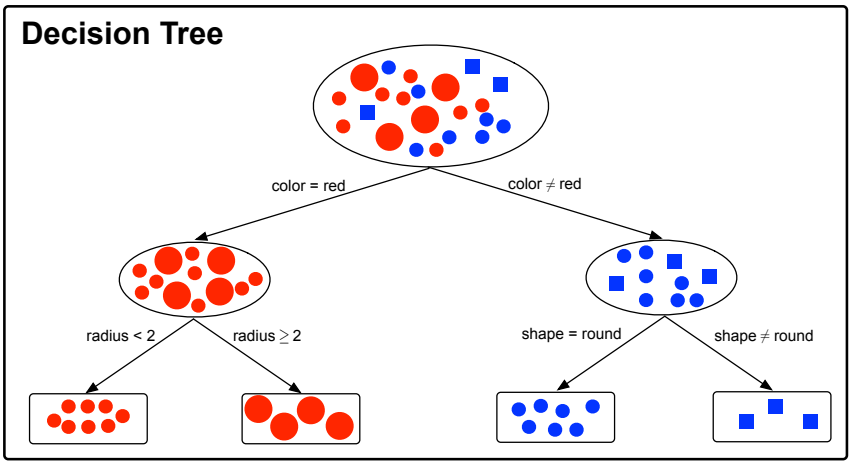

Figure 1.4: A schematic layout of a classification decision tree. Blue and red shapes are training instances in four classes (small red dot, large red dot, small blue dot and small blue rectangle). The black circles are decision tree internal nodes, and rectangles are leaf nodes. At each internal node, the decision tree partitions the input space by one of the features and a corresponding splitting value (*e.g.* radius < 2). Leaf nodes make predictions based on the instances in each leaf.

Figure 1.4 illustrates a schematic layout of a classification decision tree. Blue and red shapes are training instances in four classes (small red dot, large red dot, small blue dot and small blue rectangle). The black circles are decision tree internal nodes, and rectangles are leaf nodes. At each internal node, the decision tree partitions the input space by one of the features and a corresponding splitting value (*e.g.* radius < 2). Leaf nodes make predictions based on the instances in each leaf.

In this thesis, we focus on GBRT, a tree-based ensemble classifier. Given a continuous and differentiable function ℓ, GBRT learns an additive classifier,

$$H(\mathbf{x}) = \sum_{t=1}^{m} \eta_t h_t(\mathbf{x}), \tag{1.9}$$

where $H(\mathbf{x})$ minimizes the loss function, $h_t \in \mathcal{H}$ is one weak learner, and m is the total number of weak learners. Specifically in GBRT, each h_t is a limited depth regression tree [7] added to the current classifier at iteration t, with learning rate $\eta_t \geq 0$. $\mathcal{H}$ is the set of all possible regression trees of some limited depth b. Let H_{t-1} denote the current predictor, the regression tree h_t is selected to minimize the function $\ell(H_{t-1} + \eta_t h_t)$. This is achieved by approximating

the negative gradient of ℓ w.r.t. the current H_{t-1}, $-\frac{\partial \ell}{\partial H_{t-1}(\mathbf{x}_i)}$. The greedy classification and regression tree (CART) algorithm [7] is often used to find the approximation. CART generates a limited-depth regression tree $h_t \in \mathcal{H}$ by greedily minimizing an impurity function, $g : \mathcal{H} \to \mathcal{R}_0^+$. Typical choices for g are the squared loss,

$$h_t = \underset{h_t \in \mathcal{H}}{\operatorname{argmin}} \sum_i \Big(- \frac{\partial \ell}{\partial H_{t-1}(\mathbf{x}_i)} - h_t(\mathbf{x}_i) \Big)^2, \tag{1.10}$$

but other losses such as label entropy [53] are equally suitable. CART minimizes the impurity function (1.10) by building a decision tree similar to Figure 1.4. Consequently, h_t can be obtained by supplying $-\frac{\partial \ell}{\partial H_{t-1}(\mathbf{x}_i)}$ as the regression targets for all inputs $\mathbf{x}_i$ to an off-the-shelf CART implementation [120].

To generate predictions, GBRT uses prediction function (1.9). During testing, a test instance traverses each decision tree. Because the decision trees are of limited depth, and each split is a simple threshold on single feature, the evaluation cost is relatively low. Figure 1.2 shows the decision boundary of GBRT. Red shaded area indicates positive prediction of GBRT and blue shaded area indicates negative prediction. The decision boundary is also non-linear.

1.2.5 Parametric vs. nonparametric

Classifiers described above can also be divided into two groups: parametric classifier and nonparametric classifier. Parametric classifiers have specific functional forms governed by a small number of parameters whose value are to be learned from data. For example a linear classifier is parameterized by the weight vector $\mathbf{w}$. The important limitation of parametric model is that the chosen function might be a poor approximation to the true function that generates the observed data. In contrast, nonparametric classifiers make few assumptions about the form of the function. Instead, they treat training data as parameters and use them to make predictions. For example, a kernel regression classifier uses weighted outcomes of a test instance's training neighbors to generate the prediction. Another way to distinguish parametric and nonparametric classifiers is by the relation of the number of parameters and training instance size. Parametric classifiers have a fixed number of parameters, independent to the training size, whereas the number of parameters of nonparametric classifiers grows along with the size of training instances.

Parametric classifiers include linear regression, linear SVM and GBRT, and nonparametric classifiers include kernel regression, kernel SVM and random forest.

1.3 Motivation

Given that different classifiers have completely different intrinsic structures, their test-time cost is also vastly different. Therefore, we employ various strategies to deal with different learning scenarios. We first focus on the parametric classifiers, where the classifier evaluation cost is relatively low compared to the feature extraction cost. Since features are often heterogeneous, extraction time for different features is highly variable. Which features to extract and how to balance the trade-off between accuracy and feature extraction cost becomes a crucial problem. In this scenario, we employ a strategy that aims to reduce feature extraction cost.

GreedyMiser, described in Section 2.1, is a new algorithm that incorporates the feature extraction cost during training to explicitly minimize the CPU cost during testing. The algorithm proposes a novel impurity function to incorporate feature extraction cost and builds a connection to stage-wise regression (GBRT). The resulting classifier cherry-picks a few expensive expert features and many other good but inexpensive features to form a classifier, and greatly reduces the test-time cost.

We extend this strategy for reducing feature extraction cost to anytime classification, first introduced in [50]. Similar to the previous scenario, feature extraction cost dominates the test-time cost. However, unlike the previous scenario, in anytime classification, the test-time budget is explicitly *unknown* during training and testing. The classifier can be queried at any point to return the current best prediction. This may happen when the test-time budget is exhausted, the classifier is believed to be sufficiently accurate or the prediction is needed urgently (*e.g.* in time-sensitive applications such as pedestrian detection [47]). This unknown test-time budget introduces new problems, and we aim to learn a classifier that has a capability to produce accurate classifications at any possible budget.

Anytime Feature Representation Learning (AFR) [135], introduced in Section 2.2, describes a novel algorithm that explicitly addresses the problem of producing accurate classifications at

any budget. The algorithm lowers test-time feature extraction cost in the data representation rather than in the classifier. This enables us to turn conventional classifiers, in particular robust and accurate support vector machines (SVM), into test-time cost-sensitive anytime classifiers – combining the advantages of anytime learning and large-margin classification.

We also budget the feature extraction cost from a different perspective. Instead of limiting the cost for all test inputs, we extract different features for different inputs, and aim to constrain the *amortized* cost. We propose the second strategy, classification with trees and cascades. Consider the e-mail spam filtering example. Some of the messages can be filtered out just based on their sender-IP address in less than one millisecond (possibly without even tokenizing the message content). Others can be detected by simple text features. Still other spam e-mails can be detected only by examining image attachments using vision features. Since different inputs can be correctly classified by a variety of features that are most beneficial, expensive features are only extracted to classify a few inputs, and thus the *amortized* test-time cost can be reduced. Therefore, the goal is to construct a structured classifier directing different inputs to different paths, so the *amortized* test-time cost is within the budget.

Cost-sensitive Tree of Classifiers (CSTC) [134] described in Chapter 3 covers a tree structured classifier that focuses on trading-off accuracy and amortized test-time feature extraction cost. It builds a tree of classifiers, through which test inputs traverse along individual paths. Each path extracts different features and is optimized for a specific sub-partition of the input space. By only computing features for inputs that benefit from them the most, the cost-sensitive tree of classifiers can match the high accuracies of the current state-of-the-art classifiers at a small fraction of the computational cost. It also has a natural extension, Cost-Sensitive Cascade of Classifiers (CSCC), which is designed specifically for binary classification tasks with high class imbalance.

Finally, we trade-off accuracy and nonparametric classifier evaluation cost using a model compression strategy. We focus on the scenario where the classifier evaluation cost is no longer trivial compared to feature extraction cost. For example, the learned model of a nonparametric classifier could be very large when training set is large, and thus the test-time evaluation cost is substantial. Note that this is very common in real-world applications, as their training input size could scale to millions. Therefore the evaluation cost has to be

budgeted and accounted for during test-time. The goal of model compression is to compress very large nonparametric models with an explicit objective of constraining their evaluation cost from running over the budget.

Compressed Vector Machines (CVM) introduced in Chapter 4 is a post-processing algorithm that *compresses* the learned kernel support vector machine model by reducing and optimizing support vectors. The algorithm cherry-picks a small subset of support vectors using least angle regression (LARS) [38], and then *moves* this subset of support vectors to match the decision boundary formed by the full model. Since computing the kernel function of testing inputs and support vectors dominates the evaluation cost, reducing the number of support vectors greatly reduces the cost. Moreover, the decision boundary formed by these *moved* support vectors renders a relatively high prediction accuracy on testing inputs.

1.4 Some background in machine learning

In this section, we briefly discuss some useful background used throughout this thesis.

1.4.1 Boosting trick

Since the prediction function of gradient boosted regression trees (GBRT), H, is simply a linear function of each regression tree as in (1.9), regression trees can be interpreted as a non-linear transformation of the input data $\mathbf{x} \to \mathbf{h}(\mathbf{x})$, where $\mathbf{h}(\mathbf{x}_i) = [h_1(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)]^\top$, $h_t \in \mathcal{H}$. $\mathcal{H}$ is the set of all possible regression trees of some limited depth b (*e.g.* $b = 4$) and $T = |\mathcal{H}|$. We also denote $\boldsymbol{\beta}$ as the weight vector for transformed features, $H(\mathbf{x}) = \mathbf{h}(\mathbf{x})^\top \boldsymbol{\beta}$. The resulting feature space is extremely high dimensional and the weight-vector $\boldsymbol{\beta}$ is always kept to be correspondingly sparse. The above non-linear transformation is called the boosting-trick [44, 102, 22]. Because regression trees are negation closed (*i.e.* for each $h \in \mathcal{H}$ we also have $-h \in \mathcal{H}$) we assume throughout this thesis without loss of generality that $\boldsymbol{\beta} \geq 0$. Finally, we define a binary matrix $\mathbf{F} \in \{0, 1\}^{d \times T}$ in which an entry $F_{\alpha t} = 1$ if and only if the regression tree $h_t \in \mathcal{H}$ splits on feature α somewhere within its tree.

1.4.2 Gradient descent

Gradient descent is a numerical optimization method to find a local minimum of a function. It iteratively takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. Specifically, given a continuous and differentiable function $f(\mathbf{x})$ and an initial guess $\mathbf{x}_0$ for a local minimum of $f(\cdot)$, gradient descent method goes from $\mathbf{x}_0$ in a direction of the negative gradient, $-\frac{\partial f}{\partial \mathbf{x}}\big|_{\mathbf{x_0}}$ to a new point $\mathbf{x}_1$,

$$\mathbf{x}_1 = \mathbf{x}_0 - \eta \frac{\partial f}{\partial \mathbf{x}}\big|_{\mathbf{x_0}}, \tag{1.11}$$

where η is the learning rate. Gradient descent repeats this procedure and generates a sequence of points such that

$$\mathbf{x}_{d+1} = \mathbf{x}_d - \eta_d \frac{\partial f}{\partial \mathbf{x}}\big|_{\mathbf{x}_d},\ d \geq 0. \tag{1.12}$$

When η_d is small, the function value $f(\mathbf{x})$ monotonically decreases along this sequence,

$$f(\mathbf{x}_0) \geq f(\mathbf{x}_1) \geq \cdots \geq f(\mathbf{x}_d) \tag{1.13}$$

After certain iterations, the sequence $(\mathbf{x}_d)$ converges to a local minimum. For convergence proof, please see [6]. Note that the learning rate η_d can change at every iteration, and searching for the optimal learning rate can be done through *line search* [6].

1.4.3 Conjugate gradient descent

We briefly discuss the Polack-Ribiere conjugate gradient descent method [95] used by several of our algorithms. Similar to gradient descent, conjugate gradient descent is also a numerical optimization method to find a local minimum of a function $f(\cdot)$.

Figure 1.5 illustrates gradient descent (steepest descent) and conjugate gradient descent minimizing a quadratic function. The contours of the objective function are in gray scale, and the minimum is at the center represented by the darkest dot. Steepest descent (green arrows) takes many steps following the gradient direction. However, since it uses line search,

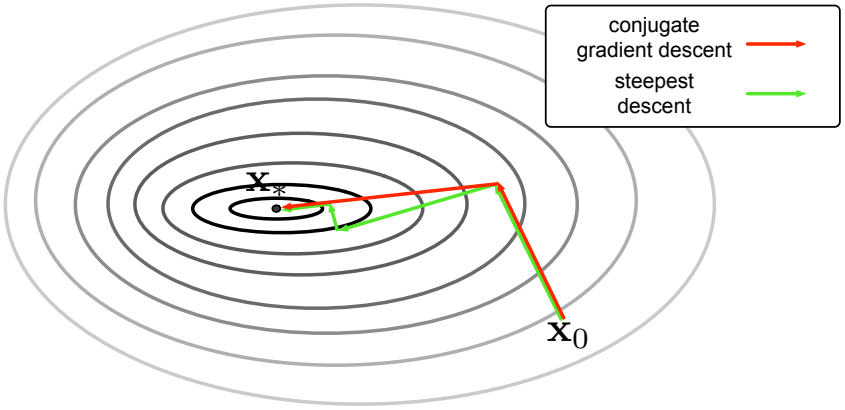

Figure 1.5: A schematic comparison of conjugate gradient descent (red) and steepest descent with line search (green).

two consecutive gradient directions are always perpendicular to each other. Therefore, even when the function is a perfectly quadratic function, steepest descent repeatedly moves along the same directions.

In contrast, conjugate gradient descent searches the descent direction based on a quadratic approximation of the function,

$$f(\mathbf{x}) \approx \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{x}^\top \mathbf{b} + c. \tag{1.14}$$

Given a positive definite matrix $\mathbf{A}$, a pair of nonzero vectors $\mathbf{p}_i, \mathbf{p}_j$ are conjugate if they are orthogonal with respect to $\mathbf{A}$,

$$\mathbf{p}_i^\top \mathbf{A}\mathbf{p}_j = 0. \tag{1.15}$$

Since every pair of conjugate directions are linearly independent, a set of d conjugate vectors spans the space in which the local minimum $\mathbf{x}_*$ lies. Using conjugate directions, we iteratively compute the next point using the update rule

$$\mathbf{x}_{d+1} = \mathbf{x}_d - \eta_d \mathbf{p}_d, \tag{1.16}$$

where $\mathbf{p}_d$ is conjugate to all previous directions $\mathbf{p}_0, \cdots, \mathbf{p}_{d-1}$, and learning rate η_d is found by line search. To initialize this iterative procedure, we use the negative gradient of function

$f(\cdot)$ at an initial guess point $\mathbf{x}_0$. In other words, $\mathbf{p}_0 = \frac{\partial f}{\partial \mathbf{x}}\big|_{\mathbf{x}_0}$. Polak [95] proves that at iteration d (the dimension of the space in which $\mathbf{x}_*$ lies), the local minimum can be found by conjugate gradient descent, $\mathbf{x}_d = \text{argmin}_{\mathbf{x}\in\mathcal{R}^d} f(\mathbf{x})$. Figure 1.5 (red arrows) shows the steps of conjugate gradient descent method. It finds the minimum with only two steps in this two dimensional space.

Chapter 2

Feature Extraction Cost Reduction

In this chapter, we focus on feature extraction cost and discuss two learning scenarios: low feature extraction cost classification and anytime classification, which employ a common strategy, reducing feature extraction cost. Section 2.1 discusses low feature extraction cost classification scenario and the proposed algorithm *Greedy Miser*. Section 2.2 discusses the second learning scenario, anytime classification and details *Anytime Feature Representation Learning (AFR)*.

2.1 Low Feature Extraction Cost Classification

Our proposed algorithm consists of many weak learners. Each weak learner is a limited depth regression tree boosted by a loss function. Different from gradient boosted regression trees (GBRT or stage-wise regression described in Section 1.2.4), our algorithm explicitly considers the test-time cost while boosting each weak learner, encouraging weak learners to cherry-pick good features and to re-use previously extracted features. We first state the (non-continuous) global objective which explicitly trades off feature extraction cost and accuracy, and relax it into a continuous loss function. Subsequently, we derive an update rule that shows the resulting loss lends itself naturally to greedy optimization with stage-wise regression [44]. Different from previous approaches [74, 105, 96, 24], our algorithm does not build cascades of classifiers. Instead, the cost/accuracy trade-off is pushed into the training and selection of the weak classifiers. The resulting learning algorithm is much simpler than any prior work, as it is a variant of regular stage-wise regression, and yet leads to superior test-time performance. We evaluate our algorithm's efficacy on two real world data sets

from very different application domains: scene recognition in images and ranking of web-search documents. Its accuracy matches that of the unconstrained baseline (with unlimited resources) while achieving an order of magnitude reduction of test-time cost.

2.1.1 Related work

Previous work on learning under test-time resource constraints appears in the context of many different applications. Most prominently, Viola and Jones [125] greedily train a cascade of weak classifiers with Adaboost [108] for visual object recognition. Cambazoglu et al. [14] propose a cascade framework explicitly for web-search ranking. They learn a set of additive weak classifiers using gradient boosting, and remove data points during test-time using proximity scores. Although their algorithm requires almost no extra training cost, the improvement is typically limited. Lefakis and Fleuret [74] and Dundar and Bi [37] learn a soft-cascade, which re-weights inputs based on their probability of passing all stages. Different from our method, they employ a global probabilistic model, do not explicitly incorporate feature extraction costs and are restricted to binary classification problems. Saberian and Vasconcelos [105] also learn classifier cascades. In contrast to prior work, they learn all cascades levels simultaneously in a greedy fashion. Unlike our approach, all of these algorithms focus on learning of cascades and none explicitly focus on individual feature costs.

To consider the feature extraction cost, Gao and Koller [45] publish an algorithm to dynamically extract features during test-time. Raykar et al. [99] learn classifier cascades, but they group features by their costs and restrict classifiers at each stage to only use a small subset. Pujara et al. [96] suggest the use of sampling to derive a cascade of classifiers with increasing cost for email spam filtering. Most recently, Chen et al. [24] introduce *Cronus*, which explicitly considers the feature extraction cost during training and constructs a cascade to encourage removal of unpromising data points early-on. At each stage, they optimize the coefficients of the weak classifiers to minimize the classification error and trees/features extraction costs. We pursue a very different (orthogonal) approach and do not optimize the cascade stages globally. Instead, we strictly incorporate the feature cost into the weak learners. Moreover, as our algorithm is a variant of stage-wise regression, it can operate naturally in both regression and multi-class classification scenarios.

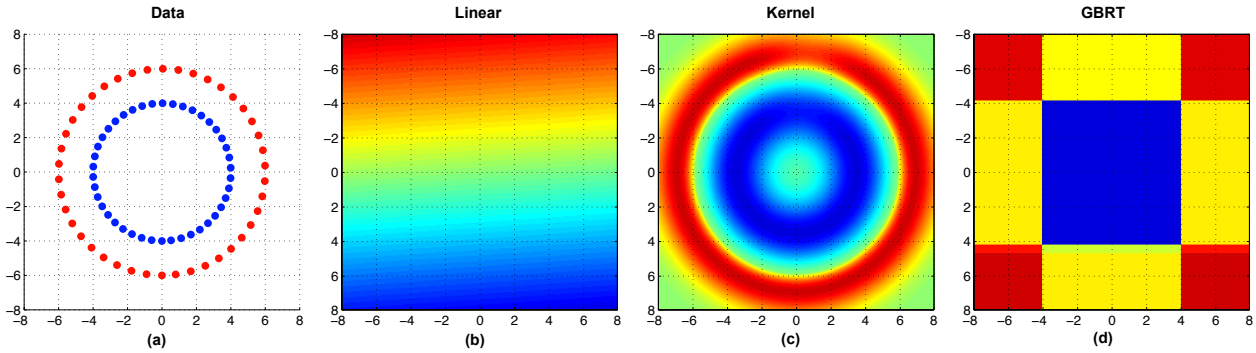

Figure 2.1: Gradient surface of a linear, a kernel and a GBRT classifier. (a) The linear un-separable simulation data. Red dots and blue dots are from two different classes, and the task is binary classification. (b) The gradient surface of a linear classifier, which is a hyper-plane (c) The gradient surface of a kernel classifier. (d) The gradient surface of a GBRT classifier.

2.1.2 Unique properties of stage-wise regression

In this subsection, we analyze the closely related algorithm of ours, stage-wise regression, and specifically, gradient boosted regression trees (GBRT) in detail. We give insights of some unique properties of GBRT that are desirable for learning under test-time resource constraints. We compare GBRT with a linear classifier and a kernel classifier. Let θ denote the parameters of a classifier H, and the goal of learning a classifier is to learn these parameters θ by minimizing a loss function ℓ. These parameters are usually learned by gradient descent, where at each iteration, we compute the gradient of the loss function ℓ w.r.t. the parameters, $\frac{\partial \ell}{\partial \theta}$. We apply the generalized chain rule, and decompose this gradient into two parts:

$$\frac{\partial \ell}{\partial \theta} = \sum_{i=1}^{n} \frac{\partial \ell}{\partial H(\mathbf{x}_i)} \frac{\partial H(\mathbf{x}_i)}{\partial \theta}, \tag{2.1}$$

where $H(\mathbf{x}_i)$ is the prediction of an input $\mathbf{x}_i$. Note that the first part $\frac{\partial \ell}{\partial H(\mathbf{x}_i)}$ is the gradient of the loss function ℓ w.r.t. predicting function in function space evaluated at an input $\mathbf{x}_i$, and the second part is the gradient of the predicting function w.r.t. its parameter θ. Compared to linear classifier and kernel classifier, there are two unique properties of GBRT: implicit parameterization and non-linear feature combination.

Implicit parameterization. Both linear classifier and kernel classifier have a predicting function H that can be represented as an analytical function of its parameters (*i.e.* for linear classifier, $H(\mathbf{x}) = \mathbf{x}^\top \theta$, where $\theta \in \mathcal{R}^d$). Therefore, when optimizing a linear or a kernel classifier, one usually optimizes the classifier parameters directly through the gradient of the loss w.r.t. classifier parameters (*i.e.* $\frac{\partial \ell}{\partial \theta}$). In contrast, there is no such an *analytical* function to model the predicting function H in GBRT. In other words, GBRT is parameterized *implicitly*, and one has to learn the parameters of GBRT predicting function in two steps. Firstly, GBRT computes the negative gradient of the loss function w.r.t. predicting function in the function space $\frac{\partial \ell}{\partial H(\mathbf{x}_i)}$ evaluated at each input sample $\mathbf{x}_i$. Secondly, GBRT approximates this negative gradient $\frac{\partial \ell}{\partial H(\mathbf{x}_i)}$ at every input $\mathbf{x}_i$ by building a limited depth regression tree $h(\cdot)$ using CART algorithm.

As described in section 1.2.4, CART generates a limited-depth regression tree $h_t \in \mathcal{H}$ by greedily minimizing an impurity function (1.10). It minimizes the impurity function by recursively splitting the data set on a single feature per tree-node. Note that features are only used when approximating the negative gradient in the CART algorithm, and therefore we can add constraints for feature extraction only in the CART algorithm. This provides more flexibility to incorporate structured feature information commonly used in computer vision, where the extraction of a feature from one vision descriptor (running one vision algorithm) sets all other features from the same descriptor free.

Non-linear feature combination. Different from linear classifiers, GBRT is capable of approximating a non-linear gradient surface. Shown in Figure 2.1 (a), we simulate a scenario where sample inputs are not linearly separable. Figure 2.1 (b-d) show the gradient surface $\frac{\partial \ell}{\partial H}$ of different classifiers. GBRT approximates the gradient surface through limited depth decision trees. For a depth 4 tree, which has $2^{4-1} = 8$ leaf nodes, GBRT approximates the gradient surface by partitioning the space into 8 blocks (shown in (d)). Kernel classifier can also achieve non-linear approximation through kernel-trick, shown in (c). In contrast, a linear classifier is just a hyperplane in the input space (*i.e.* $\mathbf{x}^\top(\frac{\partial \ell}{\partial \theta})$), which is shown in (b). The key advantage of GBRT is that while it can achieve non-linear feature combination, it is still a parametric model. During test-time, parametric GBRT is much faster than nonparametric methods such as kernel SVM, and therefore is very suitable for large scaled data sets and test-time cost-sensitive applications.

2.1.3 Greedy Miser

In this subsection, we formalize the optimization problem of test-time computational cost, and then intuitively state our algorithm. We follow the setup introduced in [24], treating stage-wise regression classifier H as a linear combination of transformed feature space $\mathbf{h}(\mathbf{x})$ using boosting trick described in Section 1.4.1.

$$H(\mathbf{x}) = \mathbf{h}(\mathbf{x})^\top \boldsymbol{\beta}, \tag{2.2}$$

where $\boldsymbol{\beta} \in \mathcal{R}^T$ is the weight vector. We then formalize the test-time computational cost of the classifier H for a given weight-vector $\boldsymbol{\beta}$.

Test-time computational cost. There are two factors that contribute to this cost: The function evaluation cost of all trees h_t with $\boldsymbol{\beta}_t > 0$ ($\boldsymbol{\beta}$ is non-negative because the set of trees is negation closed) and the feature extraction cost for all features that are used in these trees. Let $e > 0$ be the cost to evaluate one tree h_t if all features were previously extracted. Note that e is a constant independent of the number of training sample inputs, and is usually very small if the tree depth is small. This is a key advantage of stage-wise regression over other nonparametric non-linear classifiers, as stage-wise regression is a parametric classifier. With this notation and the feature extraction cost c_α and tree-feature indicator matrix $\mathbf{F}$ defined in Section 1.4.1, both costs can be expressed in a single function as

$$c(\boldsymbol{\beta}) = e\|\boldsymbol{\beta}\|_0 + \sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t=1}^{T} F_{\alpha t} \beta_t \right\|_0, \tag{2.3}$$

where the l_0-norm for scalars is defined as $\|a\|_0 \rightarrow \{0, 1\}$ with $\|a\|_0 = 1$ if and only if $a \neq 0$. The first term captures the function-evaluation cost and the second term captures the feature costs of all used features. If we combine a loss $\ell(\boldsymbol{\beta})$ with (2.3) we obtain our overall optimization problem:

$$\min_{\boldsymbol{\beta}} \ell(\boldsymbol{\beta}), \text{ subject to: } c(\boldsymbol{\beta}) \leq B, \tag{2.4}$$

where $B \geq 0$ denotes some pre-defined budget that cannot be exceeded during test-time.

Algorithm. In the remainder of this section we derive an algorithm to approximately minimize (2.4). For better clarity, we first give an intuitive overview of the resulting method in this paragraph. Our algorithm is based on stage-wise regression, which learns an additive classifier $H_{\boldsymbol{\beta}}(\mathbf{x}) = \sum_{t=1}^{m} \beta_t h_t(\mathbf{x})$ that aims to minimize the loss function (2.4).[1] During iteration t, the CART algorithm is used to generate a new tree h_t, which is added to the classifier $H_{\boldsymbol{\beta}}$.

Specifically, instead of using the regular impurity function (1.10), we propose an impurity function which on the one hand approximates the negative gradient of ℓ with the squared-loss, such that adding the resulting tree h_t minimizes ℓ, and on the other hand penalizes the initial extraction of features by their cost c_α. To capture this initial extraction cost, we define an auxiliary variable $\mathbf{u}_\alpha \in \{0, 1\}$ indicating if feature α has already been extracted ($\mathbf{u}_\alpha = 0$) in previous trees, or not ($\mathbf{u}_\alpha = 1$). We update the vector $\mathbf{u}$ after generating each tree, setting the corresponding entry for used features α to $\mathbf{u}_\alpha := 0$. If we denote s_i as the negative gradient, $s_i = -\frac{\partial \ell}{\partial H(x_i)}$, our impurity function in iteration t becomes

$$g(h_t) = \frac{1}{2} \sum_i (s_i - h_t(\mathbf{x}_i))^2 + \lambda \sum_{\alpha=1}^{d} \mathbf{u}_\alpha c_\alpha F_{\alpha t}, \tag{2.5}$$

where λ trades off the loss with the cost.

To combine the trees h_t into a final classifier $H_{\boldsymbol{\beta}}$, our algorithm follows the steps of regular stage-wise regression with a fixed step-size $\eta > 0$. As our algorithm is based on a *greedy* opti*miser*, and is stingy with respect to feature-extraction, we refer to it as *the Greedy Miser*. Algorithm (3) shows a pseudo-code implementation.

Algorithm Derivation

In this subsection, we derive a connection between (2.4) and our Greedy Miser algorithm by showing that Greedy Miser approximately solves a relaxed version of the optimization problem.

[1] Here, without loss of generality. the trees in $\mathcal{H}$ are conveniently re-ordered such that exactly the first m trees have non-zero weight β_t.

Algorithm 3 Greedy Miser in pseudo-code

Require: $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, step-size η, iterations m
  $H = 0$
  for $t = 1$ to m **do**
    $h_t \leftarrow$ Use CART to greedily minimize (2.5).
    $H \leftarrow H + \eta h_t$.
    For each feature α used in h_t, set $\mathbf{u}_\alpha \leftarrow 0$.
  end for
  Return H

Relaxation. The optimization as stated in eq. (2.4) is non-continuous, because of the l_0-norm in the cost term—and hard to optimize. We start by introducing minor relaxations to both terms in (2.3) to make it better behaved.

Assumptions. Our optimization algorithm (for details see subsection *Optimization* on page 25) performs coordinate descent and — starting from $\boldsymbol{\beta}=\mathbf{0}$ — increments one dimension of $\boldsymbol{\beta}$ by $\eta>0$ in each iteration. Because of the extremely high dimensionality (which is dictated by the number of all possible regression trees that can be represented within the accuracy of the computer) and the comparably tiny number of iterations (≤ 5000) it is reasonable to assume that one dimension is never incremented twice. In other words, the weight vector $\boldsymbol{\beta}$ is extremely sparse and (up to re-scaling by $\frac{1}{\eta}$) binary: $\frac{1}{\eta}\boldsymbol{\beta} \in \{0,1\}^T$.

Tree-evaluation cost. The l_0-norm is often relaxed into the convex and continuous l_1-norm. In our scenario, this is particularly attractive, because if $\frac{1}{\eta}\boldsymbol{\beta}$ is binary, then the re-scaled l_1 norm is identical to the l_0 norm—and the relaxation is exact. We use this approach for the first term:

$$e\|\boldsymbol{\beta}\|_0 \longrightarrow \frac{e}{\eta}\|\boldsymbol{\beta}\|_1. \tag{2.6}$$

Feature extraction cost. In the case of the feature cost, l_1-norm is not a good approximation of the original l_0-norm, because features are re-used many times, in different trees. Using l_1-norm would imply that features that are used more often would be penalized more than features that are only used once. This contradicts our assumption that features become *free* after their initial extraction.

We therefore define a new function $\|\cdot\|_{\lceil 1}$, which is a re-scaled and amputated version of the l_1-norm:

$$\|x\|_{\lceil 1} = \begin{cases} |\frac{x}{\eta}| & \text{for } |x| \in [0, \eta) \\ 1 & \text{for } |x| \in [\eta, \infty). \end{cases} \tag{2.7}$$

This penalty function $\|\cdot\|_{\lceil 1}$ behaves like the regular l_1 norm when $|x|$ is small, but is capped to a constant when $|x| \geq \eta$. Note that while we use the notation $\|\cdot\|_{\lceil 1}$ for the amputated version of the l_1-norm, it is not a well-defined norm, as it is not homogeneous ($\|tx\|_{\lceil 1} \neq t\|x\|_{\lceil 1}$) With this definition, our relaxation of the feature-cost term becomes:

$$\sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t=1}^{T} F_{\alpha t}\beta_t \right\|_0 \longrightarrow \sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t=1}^{T} F_{\alpha t}\beta_t \right\|_{\lceil 1} . \tag{2.8}$$

Similar to the previous case, if $\frac{1}{\eta}\boldsymbol{\beta}$ is binary, this relaxation is exact. This holds because in (2.8) all arguments of $\|\cdot\|_{\lceil 1}$ are non-negative multiples of η (as $F_{\alpha t} \in \{0, 1\}$ and $\beta_t \in \{0, \eta\}$) and it is easy to see from the definition of $\|\cdot\|_{\lceil 1}$ that for all $k = 0, 1, \ldots$, we have $\|k\eta\|_{\lceil 1} = \|k\eta\|_0$.

Continuous cost-term. To simplify the optimization, we split the budget into two terms $B = B_e + B_f$—the tree-evaluation budget and the feature extraction budget—and re-write (2.4) with the two penalties (2.6) and (2.8) as two individual constraints. If we use the Lagrangian formulation [6], with Lagrange multiplier λ (up to re-scaling), for the feature cost constraint and the explicit constraint formulation for the tree-evaluation cost, we obtain our final optimization problem:

$$\begin{aligned} &\min_{\boldsymbol{\beta}} \ \ell(\boldsymbol{\beta}) + \lambda \sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t} F_{\alpha t}\beta_t \right\|_{\lceil 1} \\ &\text{s.t.} \quad \frac{e}{\eta}\|\boldsymbol{\beta}\|_1 \leq B_e. \end{aligned} \tag{2.9}$$

Optimization

In this subsection we describe how Greedy Miser, our adaptation of stage-wise regression [44], finds a (local) solution to the optimization problem in (2.9).

Solution path. We follow the approach from [102] and find a solution path for (2.9) for evenly spaced tree-evaluation budgets, ranging from $B'_e = 0$ to $B'_e = B_e$. Along the path we iteratively increment B'_e by η. We repeatedly solve the intermediate optimization problem by warm-starting (2.9) with the previous solution and allowing the weight vector to change by η,

$$\begin{aligned} \min_{\boldsymbol{\delta} \geq 0} \; & \overbrace{\ell(\boldsymbol{\beta} + \boldsymbol{\delta}) + \lambda \sum_{\alpha=1}^{d} c_\alpha \left\| \sum_t F_{\alpha t}(\beta_t + \delta_t) \right\|_{\lceil 1}}^{\mathcal{L}(\boldsymbol{\beta}+\boldsymbol{\delta})}, \\ \text{s.t.} \quad & \|\boldsymbol{\delta}\|_1 \leq \eta. \end{aligned} \tag{2.10}$$

Each iteration, we update the weight vector $\boldsymbol{\beta} := \boldsymbol{\beta} + \boldsymbol{\delta}$.

Taylor approximation. The Taylor expansion of $\mathcal{L}$ is defined as

$$\mathcal{L}(\boldsymbol{\beta} + \boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\beta}) + \langle \nabla \mathcal{L}(\boldsymbol{\beta}), \boldsymbol{\delta} \rangle + O(\boldsymbol{\delta}^2). \tag{2.11}$$

If η is sufficiently small[2], and because $|\boldsymbol{\delta}| \leq \eta$, we can use the dominating linear term in (2.11) to approximate the optimization in (2.10) as

$$\min_{\boldsymbol{\delta} \geq 0} \langle \nabla \mathcal{L}(\boldsymbol{\beta}), \boldsymbol{\delta} \rangle, \;\; \text{s.t.} \;\; \|\boldsymbol{\delta}\|_1 \leq \eta. \tag{2.12}$$

Coordinate descent. The optimization (2.12) can be reduced to identifying the direction of steepest descent. Let $\nabla \mathcal{L}(\beta)_t$ denote the gradient w.r.t. the t^{th} dimension, and let us define

$$t^* = \operatorname*{argmin}_t \nabla \mathcal{L}(\beta)_t, \tag{2.13}$$

to be the gradient dimension of steepest descent. Because $\mathcal{H}$ is negation closed, we have $\nabla \mathcal{L}(\boldsymbol{\beta})_{t^*} = -\|\nabla \mathcal{L}(\boldsymbol{\beta})\|_\infty$. (If $\nabla \mathcal{L}(\boldsymbol{\beta})_{t^*} = 0$ we are done, so we focus on the case when it is < 0). With Hölder's inequality we can derive the following lower bound of the inner product

[2]Please note that we see this as a true approximation, and do not expect η to be infinitesimally small—which would cause the number of steps (and therefore trees) to become too large for practical use.

in (2.12),

$$\begin{aligned}\langle \nabla \ell(\boldsymbol{\beta}), \boldsymbol{\delta} \rangle &\geq -|\langle \nabla \mathcal{L}(\boldsymbol{\beta}), \boldsymbol{\delta} \rangle| \\ &\geq -\|\nabla \mathcal{L}(\boldsymbol{\beta})\|_\infty \|\boldsymbol{\delta}\|_1 \\ &\geq \eta \nabla \mathcal{L}(\beta)_{t^*}. \end{aligned} \tag{2.14}$$

We can now construct a vector $\boldsymbol{\delta}^*$ for which (2.14) holds as *equality*, which implies that it must be the optimal solution to (2.12). This is the case if we set $\delta^*_{t^*} = \eta$ and $\delta^*_{\neq t^*} = 0$. Consequently, we can find the solution path with steepest coordinate descent under step-size η.

Gradient derivation. The gradient $\nabla \mathcal{L}(\boldsymbol{\beta})_t$ consists of two parts, the gradient of the loss ℓ and the gradient of the feature-cost term. For the latter, we need the gradient of $\| \sum_t F_{\alpha t} \beta_t \|_{\lceil 1}$, which, according to its definition in (2.7), is not well-defined if $\sum_t F_{\alpha t} \beta_t = \eta$. As our optimization algorithm can only *increase* β_t, we derive this gradient from the *right*, yielding

$$\nabla \left\| \sum_t F_{\alpha t} \beta_t \right\|_{\lceil 1} = \begin{cases} \frac{1}{\eta} F_{\alpha t} & |\sum_t F_{\alpha t} \beta_t| < \eta \\ 0 & |\sum_t F_{\alpha t} \beta_t| \geq \eta. \end{cases} \tag{2.15}$$

Note that the condition $|\sum_t F_{\alpha t} \beta_t| < \eta$ is true if and only if feature α is not used in any trees with $\beta_t > 0$. Let us define $\mathbf{u}_\alpha = \{0, 1\}$ with $\mathbf{u}_\alpha = 1$ if and only if $|\sum_t F_{\alpha t} \beta_t| < \eta$. We can then express the gradient of $\mathcal{L}$ (with a slight abuse of notation) as

$$\nabla \mathcal{L}(\boldsymbol{\beta})_t := \frac{\partial \ell}{\partial \beta_t} + \frac{\lambda}{\eta} \sum_{\alpha=1}^{d} c_\alpha \mathbf{u}_\alpha F_{\alpha t}. \tag{2.16}$$

Applying the chain rule, we can decompose the first term in (2.16), $\frac{\partial \ell}{\partial \beta_t}$, into two parts: the derivatives w.r.t. the current prediction $H_{\boldsymbol{\beta}}(\mathbf{x}_i)$, and the partial derivatives of $H_{\boldsymbol{\beta}}(\mathbf{x}_i)$ w.r.t. β_t. This results in

$$\nabla \mathcal{L}(\boldsymbol{\beta})_t = \sum_{i=1}^{n} \frac{\partial \ell}{\partial H_{\boldsymbol{\beta}}(\mathbf{x}_i)} \frac{\partial H_{\boldsymbol{\beta}}(\mathbf{x}_i)}{\partial \beta_t} + \frac{\lambda}{\eta} \sum_{\alpha=1}^{d} c_\alpha \mathbf{u}_\alpha F_{\alpha t}. \tag{2.17}$$

As $H_{\boldsymbol{\beta}}(\mathbf{x}_i) = \mathbf{h}(\mathbf{x}_i)^\top \boldsymbol{\beta}$ is linear, we have $\frac{\partial H_{\boldsymbol{\beta}}(\mathbf{x}_i)}{\partial \beta_t} = h_t(\mathbf{x}_i)$. If we define $s_i = -\frac{\partial \ell}{\partial H_{\boldsymbol{\beta}}(\mathbf{x}_i)}$, which we can easily compute for every $\mathbf{x}_i$, we can re-phrase (2.17) as

$$\nabla \mathcal{L}(\boldsymbol{\beta})_t = \sum_{i=1}^{n} -s_i h_t(\mathbf{x}_i) + \frac{\lambda}{\eta} \sum_{\alpha=1}^{d} c_\alpha \mathbf{u}_\alpha F_{\alpha t}. \tag{2.18}$$

The Greedy Miser. For simplicity, we restrict $\mathcal{H}$ to only normalized regression-trees (*i.e.* $\sum_i h_t^2(\mathbf{x}_i) = 1$), which allows us to add two constant terms $\frac{1}{2}\sum_i h_t^2(\mathbf{x}_i)$ and $\frac{1}{2}\sum_i s_i^2$ to (2.18) without affecting the outcome of the minimization in (2.13), as both are independent of t. This completes the binomial equation and we obtain a quadratic form:

$$h_t = \operatorname*{argmin}_{h_t \in \mathcal{H}} \frac{1}{2} \sum_{i}^{n} (s_i - h_t(\mathbf{x}_i))^2 + \lambda' \sum_{\alpha=1}^{d} c_\alpha \mathbf{u}_\alpha F_{\alpha t}, \tag{2.19}$$

with $\lambda' = \frac{\lambda}{\eta}$. Note that (2.19) is exactly what Greedy Miser minimizes in (2.5), which concludes our derivation.

Meta-parameters. The meta-parameters of Greedy Miser are surprisingly intuitive. The maximum number of iterations, m, is tightly linked to the tree-evaluation budget B_e. The optimal solution of (2.12) must satisfy the *equality* $\|\boldsymbol{\delta}^*\|_1 = \eta$ (unless $\nabla \mathcal{L} = \mathbf{0}$, in which case a local minimum has been reached and the algorithm would terminate). As $\|\boldsymbol{\beta}\|_1$ is exactly increased by η in each iteration, it can be expressed in terms of the number of iterations m of the algorithm, and we obtain $\frac{1}{\eta}\|\boldsymbol{\beta}\|_1 = m$. Consequently, in order to satisfy the l_1 constraint in (2.9), we must limit to the number of iterations to $m \leq \frac{B_e}{e}$. The parameter λ' corresponds directly to the feature-budget B_f. The algorithm is not particularly sensitive to the exact step-size η, and throughout the results section we set it to $\eta = 0.1$.

2.1.4 Results

We conduct experiments on two learning under test-time resource constraints benchmark tasks from very different domains: the Yahoo Learning to Rank Challenge data set [18] and the scene recognition data set from [71].

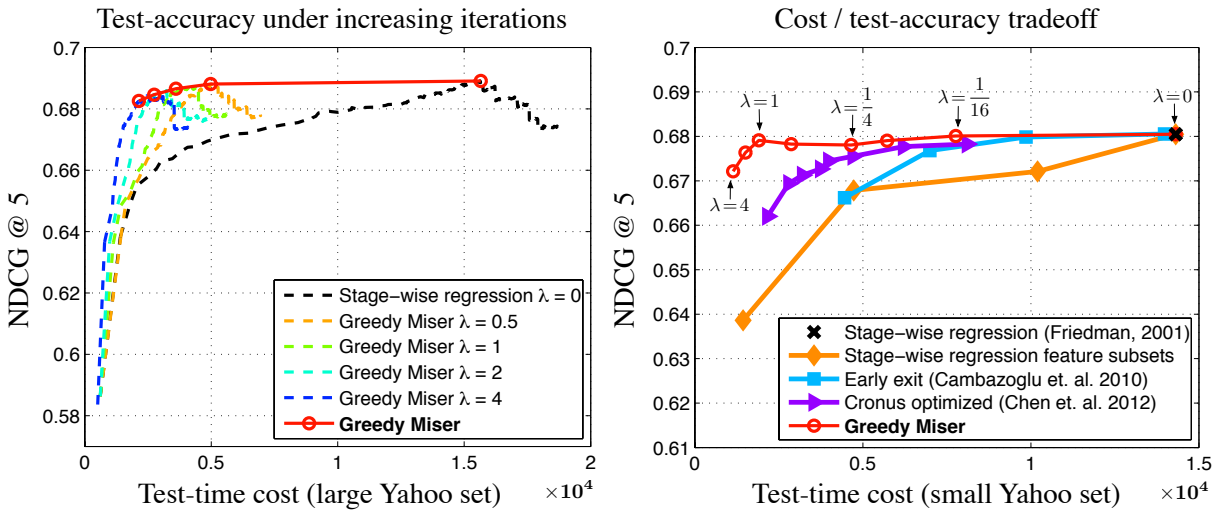

Figure 2.2: The NDCG@5 and the test-time cost of various classifier settings. *Left:* The comparison of the original *Stage-wise regression* ($\lambda = 0$) and Greedy Miser under various feature-cost/accuracy trade-off settings (λ) on the full Yahoo set. The dashed lines represent the NDCG@5 as trees are added to the classifier. The red circles indicate the best scoring iteration on the validation data set. *Right:* Comparisons with prior work on test-time optimized cascades on the small Yahoo set. The cost-efficiency curve of Greedy Miser is consistently above prior work, reducing the cost, at similar ranking accuracy, by a factor of 10.

Yahoo Learning to Rank. The Yahoo data set contains document/query pairs with label values from $\{0, 1, 2, 3, 4\}$, where 0 means the document is irrelevant to the query, and 4 means highly relevant. In total, it has 473134, 71083, 165660, training, validation, and testing pairs. As this is a regression task, we use the squared-loss as our loss function ℓ. Although the data set is representative for a web-search ranking *training* data set, in a real world *test* setting, there are many more irrelevant data points. Usually, for each query, only a few documents are relevant, and the other hundreds of thousands are completely irrelevant. Therefore, we follow the convention of [24] and replicate each irrelevant data point (label value is 0) 10 times.

Each feature in the data set has an acquisition cost. The feature costs are discrete values in the set $\{1, 5, 10, 20, 50, 100, 150\}$. The unit of these costs is approximately the time to evaluate a tree $h_t(\cdot)$. The cheapest features (cost value is 1) are those that can be acquired by looking up a table (such as the statistics of a given document), whereas the most expensive ones (such as BM25F-SD described in [9]), typically involve term proximity scoring.

To evaluate the performance on this task, we follow the typical convention and use Normalized Discounted Cumulative Gain (NDCG@5) [58], as it places stronger emphasis on retrieving relevant documents within a large set of irrelevant documents. Let π be an ordering of all inputs associated with a particular query ($\pi(z)$ is the index of the z^{th} ranked document and $y_{\pi(r)}$ is its relevance label), then the NDCG of π at position P is defined as

$$NDCG@P(\pi) = \frac{DCG@P(\pi)}{DCG@P(\pi^*)} \text{ with } DCG@P(\pi) = \sum_{z=1}^{P} \frac{2^{y_{\pi(z)}} - 1}{\log_2(z+1)}, \tag{2.20}$$

where π^* is an optimal ranking (*i.e.* documents are sorted in decreasing order of relevance).

Loss/cost trade-off. Figure 2.2 (*left*) shows the traces (dashed lines) of the NDCG@5/cost generated by repeatedly adding trees to the predictor until 3000 trees in total — essentially depicting the results under increasing tree-evaluation budgets B_e. The different traces are obtained under varying values of the feature-cost trade-off parameter λ. The baseline, *stage-wise regression* [44], is equivalent to Greedy Miser with $\lambda = 0$ and is essentially building trees without any cost consideration. The red circles indicate the iteration with the highest NDCG@5 value on the validation data set. The graph shows, that under increased λ (the solid red line), the NDCG@5 ranking accuracy of Greedy Miser drops very gradually, while the test-time cost is reduced drastically (compared to $\lambda=0$).

Comparison with prior work. In addition to stage-wise regression, we also compare against *Stage-wise regression feature subsets*, *Early Exit* [14] and *Cronus* [24]. *Stage-wise regression feature subsets* is a natural extension to stage-wise regression. We group all features according to the feature cost, and gradually use more expensive feature groups. The curve is generated by only using features whose cost $\leq 1, 20, 100, 200$. *Early Exit*, proposed by [14], trains trees identical to stage-wise regression—however, it reduces the average test-time cost by removing unpromising documents early-on during test-time. Among all methods of early-exit the authors suggested, we plot the best performing one (Early Exit Using Proximity Threshold). We introduce an early exit every 10 trees (300 in total), and at the i^{th} early-exit, we remove all test-inputs that have a score of at least $\frac{(300-i)s}{299}$ lower than the fifth best input (where s is a parameter regulating the pruning aggressiveness). The overall improvement over stage-wise regression is limited because the cost is dominated by the feature acquisition, rather than tree computation. It is worth pointing out that the

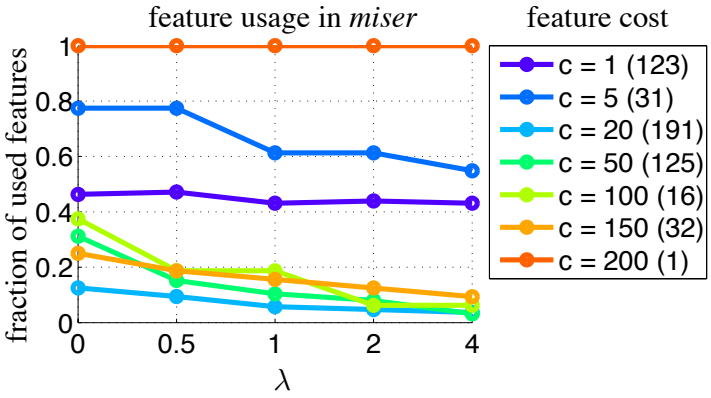

Figure 2.3: Features (grouped by cost c) used in Greedy Miser with various λ (the number of features in each cost group is indicated in parentheses in the legend). Most cheap features ($c=1$) are extracted constantly in different λ settings, whereas expensive features ($c\geq 5$) are extracted more often when λ is small. The most expensive (and invaluable) feature $c = 200$ is always extracted.

cascade-based approaches of Early-Exits and Cronus are actually complementary to Greedy Miser.

Since *Cronus* does not scale to the full data set, we use the subset of the Yahoo data from [24] of 141397, 146769, 184968, training, validation and testing points respectively. In comparison to *Cronus*, which requires $O(mn)$ memory, Greedy Miser requires no significant operational memory besides the data and scales easily to millions of data points. Figure 2.2 (*right*) depicts the trade-off curves, of Greedy Miser and competing algorithms, between the test-time cost and generalization error. We generate the curves by varying the feature-cost trade-off λ (or the pruning parameter s for *Early-Exits*). For each setting we choose the iteration that has the best validation NDCG@5 score. The graph shows that all algorithms manage to match the unconstrained cost-results of stage-wise regression. However, the trade-off curve of Greedy Miser stays consistently above that of *Cronus* and *Early Exits*, leading to better ranking accuracy at lower test-time cost. In fact, Greedy Miser can almost match the ranking accuracy of stage-wise regression with 1/10 of the cost, whereas *Cronus* reduces the cost only to 1/4 and *Early-Exits* to 1/2.

Feature extraction. To investigate what effect the feature-cost trade-off parameter λ has on the classifier's feature choices, Figure 2.3 visualizes what type of features are extracted by Greedy Miser as λ increases. For this visualization, we group features by cost and show

Figure 2.4: Sample images of the Scene 15 classification task.

what fraction of features in each group are extracted. The legend in the right indicates the cost of a feature group and the number of features that fall into it (in the parentheses). We plot the feature fraction at the best performing iteration based on the validation set. With $\lambda = 0$, Greedy Miser does not consider the feature cost when building trees, and thus extracts a variety of expensive features. As λ increases, it extracts fewer expensive features and reuses more cheap features ($c_\alpha = 1$). It is interesting to point out that across all different Greedy Miser settings, a few expensive features (cost ≥ 150) are always extracted within early iterations. This highlights a great advantage of Greedy Miser over some other cascade algorithms [99], which learn cascades with pre-assigned feature costs and cannot extract good but expensive features until the very end.

Scene Recognition. The Scene-15 data set [71] is from a very different data domain. It contains 4485 images from 15 scene classes and the task is to classify images according to scene. Figure 2.4 shows one example image for each scene category. We follow the procedure used by [71, 76], randomly sampling 100 images from each class, resulting in 1500 training images. From the remaining 2985 images, we randomly sample 20 images from each class as validation, and leave the rest 2685 for test.

We use a diverse set of visual descriptors varying in computation time and accuracy: GIST, spatial HOG, Local Binary Pattern, self-similarity, texton histogram, geometric texton, geometric color, and Object Bank [76]. The authors from Object Bank apply 177 object detectors to each image, where each object detector works independently of each other. We treat each object detector as an independent descriptor and end up with a total of 184 different visual descriptors.

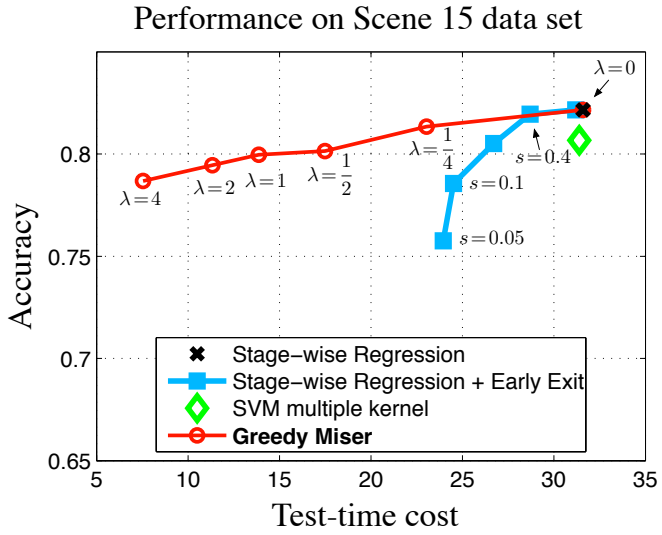

Figure 2.5: Accuracy as a function of CPU-cost during test-time. The curve is generated by gradually increasing λ. Greedy Miser champions the accuracy/cost trade-off and obtains similar accuracy as the SVM with multiple kernels with only half its test-time cost.

We split the training data 30/70 and use the smaller subset to construct a kernel and train 15 one-vs-all SVMs for each descriptor. We use the predictions of these SVMs on the larger subset as the features of Greedy Miser (totaling $d = 184 \times 15 = 2760$ features.) As for loss function ℓ, we use the multi-class log-loss [53] and maintain 15 tree-ensemble classifiers $H^1, \ldots, H^{15}$, one for each class. During each iteration, we construct 15 regression trees (depth 3) and update all classifiers. For a given image, each classifier's (normalized exponential) output represents the probability of this data point belonging to one class.

We compute the feature-extraction-cost as the CPU-time required for the computation for the visual descriptor, the kernel construction and the SVM evaluation. Each visual descriptor is used by 15 one-vs-all features. The moment any one of these features is used, we set the feature extraction cost of all other features that are based on the same visual descriptor to only the SVM evaluation time (*e.g.* if the first HOG-based feature is used, the cost of all other HOG-based features is reduced to the time required to evaluate the SVM). This is a key advantage of Greedy Miser, which inherits the two-steps optimization procedure from GBRT. The feature extraction is handled by CART algorithm and its impurity function (2.5). Therefore, Greedy Miser can easily handle structured feature extraction information by arbitrarily setting the feature cost identity variable $\mathbf{u}_\alpha$ at *any* iteration. In this example, by

setting the $\mathbf{u}_\alpha = 0$ for all HOG-based features after extracting the first HOG-based feature, the CART algorithm is encouraged to build trees using HOG-based features exclusively until starting to see diminishing rewards.

Figure 2.5 summarizes the results on the Scene-15 data set. As baseline we use stage-wise regression [44] and an SVM with the averaged kernel of all descriptors. We also apply stage-wise regression with *Early Exits*. As this is multi-class classification instead of regression we introduce an early exit every 10 trees (300 in total), and we remove test-inputs whose maximum class-likelihood is greater than a threshold s. We generate the curve of early exit by gradually increasing the value for s. The last baseline is original vision features with l_1 regularization, and we notice that its accuracy never exceeds 0.74, and therefore we do not plot it. The Greedy Miser curve is generated by varying loss/feature-cost trade-off λ. For each setting we choose the iteration that has the best validation accuracy, and all results are obtained by averaging over 10 randomly generated training/testing splits.

Both multiple-kernel SVM and stage-wise regression achieve high accuracy, but their need to extract all features significantly increases their cost. Early Exit has only limited improvement due to the inability to select a few expensive but important features in early iterations. As before, Greedy Miser champions the cost/accuracy trade-off and its accuracy drops gently with increasing λ.

All experiments (on both data sets) were conducted on a desktop with dual 6-core Intel i7 cpus with 2.66GHz. The training time for Greedy Miser requires comparable amount of time as stage-wise regression (about 80 minutes for the full Yahoo data set and 12 minutes for Scene-15.)

2.1.5 Conclusion

Greedy Miser focuses on reducing the test-time feature extraction cost. It is closely connected to stage-wise regression, and explicitly incorporates feature extraction cost. It is simple to implement, naturally scales to large data sets and outperforms previously most cost-effective classifiers. Greedy Miser can also be naturally extended to feature selection problems [136] by setting a uniform cost for all features.

2.2 Anytime Classification

In *anytime classification* setting [50], classifiers extract features on-demand during test-time and can be queried at any point to return the current best prediction. This may happen when the cost budget is exhausted, the classifier is believed to be sufficiently accurate or the prediction is needed urgently (*e.g.* in time-sensitive applications such as pedestrian detection [47]). Different from previous settings in learning under test-time resource constraints, the cost budget is explicitly *unknown* during test-time.

In this section, we address this setting with a novel approach to learning under test-time resource constraints. In contrast to most previous work, we learn an additive anytime *representation*. During test-time, an input is mapped into a feature space with multiple stages: each stage refines the data representation and is accompanied by its own SVM classifier, but adds extra cost in terms of feature extraction. We show that the SVM classifiers and the cost-sensitive anytime representations can be learned *jointly* in a single optimization.

Our method, Anytime Feature Representation Learning (AFR), is the first to incorporate anytime learning into large margin classifiers—combining the benefits of both learning frameworks. On two real world benchmark data sets *anytime* AFR out-performs or matches the performance of the current state-of-the-art learning under test-time resource constraints algorithms which are trained with a *known* test-time budget.

2.2.1 Related work

Prior work addresses anytime classification primarily with additive ensembles, obtained through boosted classifiers [125, 49]. Here, the prediction is refined through an increasing number of weak learners and can naturally be interrupted at any time to obtain the current classification estimate. Anytime adaptations of other classification algorithms where early querying of the evaluation function is not as natural—such as the popular SVM—have until now remained an open problem.

Grubb and Bagnell [49] combine gradient boosting and neural networks through back-propagation. Their approach shares a similar structure with our algorithm, as AFR can

be regarded as a two layer neural network, where the first layer is non-linear decision trees and the second layer a large margin classifier. However, different from ours, their approach focuses on avoiding local minima and does not aim to reduce test-time cost.

2.2.2 Background

Our algorithm consists of two jointly integrated parts, classification and representation learning. For the former we use support vector machines (SVM) [29] and for the latter we use the Greedy Miser [132] described in Section 2.1. In the following, we re-cap these two algorithms.

Support vector machines (SVM). Let ϕ denote a mapping that transforms inputs $\mathbf{x}_i$ into feature vectors $\phi(\mathbf{x}_i)$. Further, we define a weight vector $\mathbf{w}$ and bias b. SVMs learn a maximum margin separating hyperplane by solving a constrained optimization problem,

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\mathbf{w}^\top\mathbf{w} + \frac{1}{2}C\sum_i^n [1 - y_i(\mathbf{w}^\top\phi(\mathbf{x}_i) + b)]_+^2, \tag{2.21}$$

where constant C is the regularization trade-off hyper-parameter, and $[a]_+ = \max(a, 0)$. The squared hinge-loss penalty guarantees differentiability of (2.21), and simplifies the derivation in Section 2.2.3. A test input is classified by the sign of the SVM predicting function

$$J[\phi(\mathbf{x}_j)] = \mathbf{w}^\top\phi(\mathbf{x}_j) + b. \tag{2.22}$$

Greedy Miser. Introduced in Section 2.1, Greedy Miser incorporates feature cost into gradient boosting. Let $c_f(H)$ denote the test-time feature extraction cost of a gradient boosted tree ensemble H and $c_e(H)$ denote the CPU time to evaluate all trees[3]. Let $B_f, B_e > 0$ be corresponding finite cost budgets. The Greedy Miser solves the following optimization problem:

$$\min_H \ell(H), \text{ s.t. } c_e(H) \leq B_e \text{ and } c_f(H) \leq B_f, \tag{2.23}$$

[3]Note that both costs can be in different units. Also, it is possible to set $c_e(H) = 0$ for all H. We set the evaluation cost of a single tree to 1 cost unit.

where ℓ is continuous and differentiable. To formalize the *feature cost*, an auxiliary function $\mathbf{u}_\alpha(h_t) \in \{0, 1\}$ is defined, indicating if feature α is used in tree h_t for the first time, (*i.e.* $\mathbf{u}_\alpha(h_t) = 1$). In Section 2.1, it has been shown that by incrementally selecting h_t according to

$$\min_{h_t \in \mathcal{H}} \sum_i \Big(-\frac{\partial \ell}{\partial H_{t-1}(\mathbf{x}_i)} - h_t(\mathbf{x}_i) \Big)^2 + \lambda \sum_\alpha \mathbf{u}_\alpha(h_t) c_\alpha, \tag{2.24}$$

the constrained optimization problem in (2.23) is (approximately) minimized up to a local minimum (stronger guarantees exist if ℓ is convex). Here, λ trades off the classification loss with the feature extraction cost (enforcing budget B_f) and the maximum number of iterations limits the tree evaluation cost (enforcing budget B_e).

2.2.3 Anytime feature representation

As a lead-up to anytime feature representations, we formulate the learning of the feature representation mapping $\phi : \mathcal{R}^d \rightarrow \mathcal{R}^S$ and the SVM classifier $(\mathbf{w}, b)$ such that the costs of the final classification $c_f(J[\phi(\mathbf{x})]), c_e(J[\phi(\mathbf{x})])$ are within cost budgets B_f, B_e. In the following section we extend this formulation to an anytime setting, where B_f and B_e are *unknown* and the user can interrupt the classifier at any time. As the SVM classifier is *linear*, we consider its evaluation free during test-time and the cost c_e originates entirely from the computation of $\phi(\mathbf{x})$.

Boosted representation. We learn a representation with a variant of the boosting trick [118, 23]. To differentiate the original features $\mathbf{x}$ and the new feature representation $\phi(\mathbf{x})$, we refer only to original features as "*features*", and the components of the new representation as "*dimensions*". In particular, we learn a representation $\phi(\mathbf{x}) \in \mathcal{R}^S$ through the mapping function ϕ, where S is the total number of dimensions of our new representation. Each dimension s of $\phi(\mathbf{x})$ (denoted $[\phi]^s$) is a gradient boosted classifier, *i.e.* $[\phi]^s = \eta \sum_{t=0}^{T} h_t^s$. Specifically, each h_t^s is a limited depth regression tree.

For each dimension s, we initialize $[\phi]^s$ with the s^{th} tree obtained from running the Greedy Miser for S iterations with a very small feature budget B_f. Subsequent trees are learned as

described in the following. During classification, the SVM weight vector $\mathbf{w}$ assigns a weight w_s to each dimension $[\phi]^s$.

Train/Validation Split. As we learn the feature representation ϕ *and* the classifier $\mathbf{w}, b$ jointly, overfitting is a concern, and we carefully address it in our learning setup. Usually, overfitting in SVMs can be overcome by setting the regularization trade-off parameter C carefully with cross-validation. In our setting, however, the representation *changes* and the hyper-parameter C needs to be adjusted correspondingly. We suggest a more principled setup, inspired by Chapelle et al. [20], and also learn the hyper-parameter C. To avoid trivial solutions, we divide our training data into two equally-sized parts, which we refer to as training and validation sets, $\mathcal{T}$ and $\mathcal{V}$. The representation is learned on both sets, whereas the classifier $\mathbf{w}, b$ is trained only on $\mathcal{T}$, and the hyper-parameter is tuned for $\mathcal{V}$. We further split the validation set into validation $\mathcal{V}$ and a held-out set $\mathcal{O}$ in a 80/20 split. The held-out set $\mathcal{O}$ is used for early-stopping.

Nested optimization. We define a loss function that approximates the 0-1 loss on the validation set $\mathcal{V}$,

$$\ell_{\mathcal{V}}(\phi; \mathbf{w}, b) = \sum_{\mathbf{x}_i \in \mathcal{V}} \mu_{y_i} \sigma\Big(J(\phi(\mathbf{x}_i))\Big), \tag{2.25}$$

where $\sigma(z) = \frac{1}{1+e^{az}}$ is a soft approximation of the $sign(\cdot)$ step function (we use $a = 5$ throughout, similar to [20, 133]) and $\mu_{y_i} > 0$ denotes a class specific weight to address potential class imbalance. $J(\cdot)$ is the SVM predicting function defined in (2.22). The classifier parameters $(\mathbf{w}, b)$ are assumed to be the optimal solution of (2.21) for the training set $\mathcal{T}$. We can express this relation as a nested optimization problem (in terms of the SVM parameters $\mathbf{w}, b$) and incorporate our test-time budgets B_e, B_f:

$$\begin{aligned} &\min_{\phi, C} \ \ell_{\mathcal{V}}(\phi, \mathbf{w}, b), \text{ s.t. } c_e(\phi) \leq B_e \text{ and } c_f(\phi) \leq B_f \\ &\quad \min_{\mathbf{w}, b} \ \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2} C \sum_i^n \mu_{y_i} [1 - y_i(\mathbf{w}^\top \phi(\mathbf{x}_i) + b)]_+^2. \end{aligned} \tag{2.26}$$

According to Theorem 4.1 in [5], $\ell_{\mathcal{V}}$ is continuous and differentiable based on the uniqueness of the optimal solution $\mathbf{w}^*, b^*$. This is a sufficient prerequisite for being able to solve $\ell_{\mathcal{V}}$ via the Greedy Miser (2.24), and since the constraints in (2.26) are analogous to (2.23), we can optimize it accordingly.

Algorithm 4 AFR in pseudo-code.

Initialize $\lambda = \lambda_0$, $s_0 = 1$
while $\lambda > \epsilon$ **do**
  Initialize $\phi = [h_0^{s_0}(\cdot), \ldots, h_0^{s_0+S}(\cdot)]^\top$ with (2.24).
  for $s = s_0$ **to** $s_0 + S$ **do**
    for $t = 1$ **to** T **do**
      Train an SVM using ϕ to obtain $\mathbf{w}$ and b.
      If accuracy on $\mathcal{O}$ has increased, **continue**.
      Compute gradients $\frac{\partial \ell_{\mathcal{V}}}{\partial [\phi]^s}$ and $\frac{\partial \ell_{\mathcal{V}}}{\partial C}$
      Update $C = C - \gamma \frac{\partial \ell_{\mathcal{V}}}{\partial C}$
      Call CART with impurity (2.27) to obtain h_t^s
      Stop if $\sum_i h_t^s(\mathbf{x}_i) \frac{\partial \ell_{\mathcal{V}}}{\partial [\phi]^s(\mathbf{x}_i)} < 0$
      Update $[\phi]^s = [\phi]^s + \eta h_t^s$.
    end for
  end for
  $\lambda := \lambda/2$ and $s_0 += S$.
end while

Tree building. The optimization (2.26) is essentially solved by a modified version of gradient descent, updating ϕ and C. Specifically, for fast computation, we update one dimension $[\phi]^s$ at a time, as we can utilize the previous learned tree in the same dimension to speed up computation for the next tree [120]. The computation of $\frac{\partial \ell_{\mathcal{V}}}{\partial [\phi]^s}$ and $\frac{\partial \ell_{\mathcal{V}}}{\partial C}$ is described in detail in subsection *Optimization* in page 41. At each iteration, the tree h_t^s is selected to trade-off the gradient fit of the loss function $\ell_{\mathcal{V}}$ with the feature cost of the tree,

$$\min_{h_t^s} \sum_i \Big(- \frac{\partial \ell_{\mathcal{V}}}{\partial [\phi]^s(\mathbf{x}_i)} - h_t^s(\mathbf{x}_i) \Big)^2 + \lambda \sum_\alpha \mathbf{u}_\alpha(h_t^s) c_\alpha. \tag{2.27}$$

We use the learned tree h_t^s to update the representation $[\phi]^s = [\phi]^s + \eta h_t^s$. At the same time, the variable C is updated with small gradient steps.

Anytime feature representations

Minimizing (2.26) results in a cost-sensitive SVM $(\mathbf{w}, b)$ that uses a feature representation $\phi(\mathbf{x})$ to make classifications within test-time budgets B_f, B_e. In the anytime learning setting, however, the test-time budgets are *unknown*. Instead, the user can interrupt the test evaluation at any time.

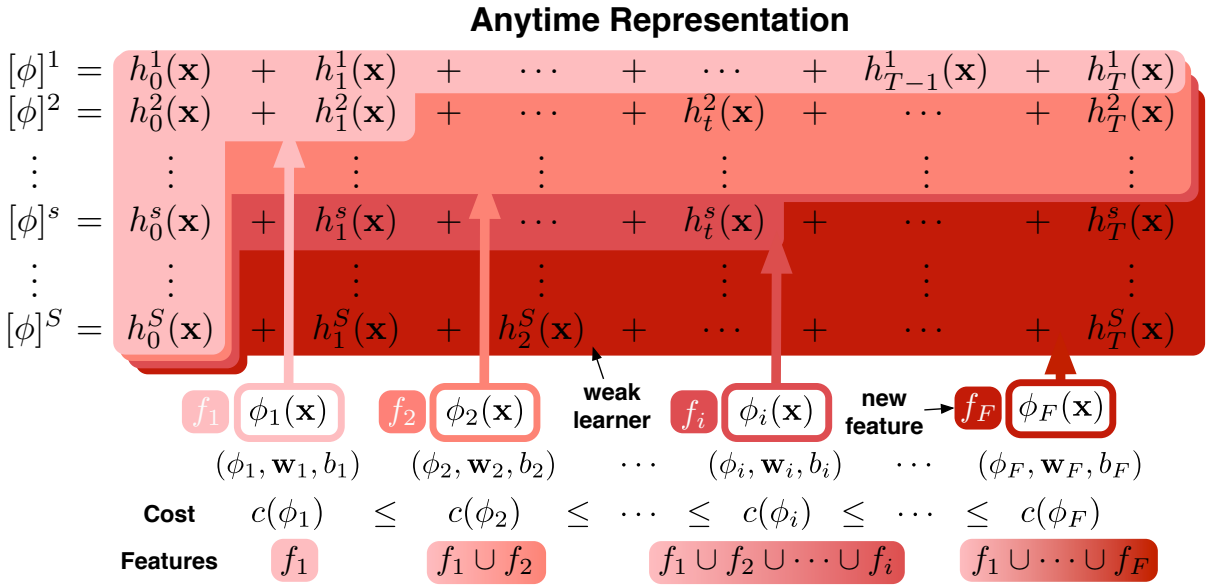

Figure 2.6: A schematic layout of Anytime Feature Representation Learning. Different shaded areas indicate representations of different costs, the darker the costlier. During training time, SVM parameters $\mathbf{w}, b$ are saved every time a new feature f_i is extracted. During test-time, under budgets B_e, B_f, we use the most expensive triplet $(\phi_k, \mathbf{w}_k, b_k)$ with cost $c_e(\phi_k) \leq B_e$ and $c_f(\phi_k) \leq B_f$.

Anytime parameters. We refer to our approach to anytime setting as *Anytime Feature Representations* (AFR) and Algorithm 4 summarizes the individual steps of AFR in pseudo-code. We obtain an anytime setting by steadily increasing B_e and B_f until the cost constraint has no effect on the optimal solution. In practice, the tree budget (B_e) increase is enforced by adding one tree h_t^s at a time (where t ranges from 1 to T). The feature budget B_f is enforced by the parameter λ in (2.27). As the feature cost is dominant, we slowly decrease λ (starting from some high value λ_0). For each intermediate value of λ we learn S dimensions of $\phi(\mathbf{x})$ (each dimension consisting of T trees). Whenever all S dimensions are learned, λ is divided by a factor of 2 and an additional S dimensions of $\phi(\mathbf{x})$ are learned and concatenated to the existing representation.

Whenever a new feature is extracted by a tree h_t^s, the cost increases substantially. Therefore we store the learned representation mapping function and the learned SVM parameters whenever a new feature is extracted. We overload ϕ_α to denote the representation learned with feature α^{th} extracted, and $\mathbf{w}_\alpha, b_\alpha$ as the corresponding SVM parameters. Storing these

parameters results in a series of triplets $(\phi_1, \mathbf{w}_1, b_1) \dots (\phi_F, \mathbf{w}_F, b_F)$ of increasing cost, *i.e.* $c(\phi_1) \leq \dots \leq c(\phi_F)$ (where F is the total number of extracted features). Note that we save the mapping function ϕ, rather than the representation of each training input $\phi(\mathbf{x})$.

Evaluation. During test time, the classifier may be stopped during the extraction of the $\alpha + 1^{th}$ feature, because the feature budget B_f (unknown during training time) has been reached. In this case, to make a prediction, we sum the previously-learned representations generated by the first α features $\mathbf{w}_\alpha^\top \sum_{k=1}^{\alpha} \phi_k(\mathbf{x}) + b_\alpha$. This approach is schematically depicted in Figure 2.6.

Early-stopping. Updating each dimension with a fixed number of T trees may lead to overfitting. We apply early-stopping by evaluating the prediction accuracy on the hold-out set $\mathcal{O}$. We stop adding trees to each dimension whenever this accuracy decreases. Algorithm (4) details all steps of our algorithm.

Optimization

Updating feature representation $\phi(\mathbf{x})$ requires computing the gradient of the loss function $\ell_\mathcal{V}$ w.r.t. $\phi(\mathbf{x})$ as stated in (2.27). In this subsection we explain how to compute the necessary gradients efficiently.

Gradient w.r.t. $\phi(\mathbf{x})$. We use the chain rule to compute the derivative of $\ell_\mathcal{V}$ w.r.t. each dimension $[\phi]^s$,

$$\frac{\partial \ell_\mathcal{V}}{\partial [\phi]^s} = \frac{\partial \ell_\mathcal{V}}{\partial J} \frac{\partial J}{\partial [\phi]^s}, \tag{2.28}$$

where J is the prediction function in (2.22). As changing $[\phi]^s$ not only affects the validation data, but also the representation of the training set, $\mathbf{w}$ and b are also functions of $[\phi]^s$. The derivative of J w.r.t. the representation of the training inputs, $[\phi]^s \in \mathcal{T}$ is

$$\frac{\partial J}{\partial [\phi]^s} = \Big(\frac{\partial \mathbf{w}}{\partial [\phi]^s} \Big)^\top \phi_\mathcal{V} + \frac{\partial b}{\partial [\phi]^s}, \tag{2.29}$$

where we denote all validation inputs by $\phi_{\mathcal{V}}$. For validation inputs, the derivative w.r.t. $[\phi]^s \in \mathcal{V}$ is

$$\frac{\partial J}{\partial [\phi]^s} = \mathbf{w}^\top \frac{\partial \phi_{\mathcal{V}}}{\partial [\phi]^s}. \tag{2.30}$$

Note that with $|\mathcal{T}|$ training inputs and $|\mathcal{V}|$ validation inputs, the gradient consists of $|\mathcal{T}|+|\mathcal{V}|$ components.

In order to compute the remaining derivatives $\frac{\partial \mathbf{w}}{\partial [\phi]^s}$ and $\frac{\partial b}{\partial [\phi]^s}$, we will express $\mathbf{w}$ and b in closed-form w.r.t. $[\phi]^s$. First, let us define the contribution to the loss of input $\mathbf{x}_i$ as $\xi_i = [1 - y_i(\mathbf{w}^{*\top}\phi(\mathbf{x}_i) + b^*)]_+$. The optimal value $\mathbf{w}^*, b^*$ is only affected by support vectors (inputs with $\xi_i > 0$). Without loss of generality, let us assume that those inputs are the first $\mathbf{sv}$ in our ordering, $\mathbf{x}_1, \ldots, \mathbf{x}_{\mathbf{sv}}$. We remove all non-support vectors, and let $\widehat{\mathbf{\Phi}} = [y_1\phi_1, \ldots, y_{n_{\mathbf{sv}}}\phi_{n_{\mathbf{sv}}}]$, and $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_{n_{\mathbf{sv}}}]^\top$. We also define a diagonal matrix $\Lambda \in \mathcal{R}^{n_{\mathbf{sv}} \times n_{\mathbf{sv}}}$ whose diagonal elements are class weight $\Lambda_{ii} = \mu_{y_i}$. We can then rewrite the nested SVM optimization problem within (2.26) in matrix form:

$$\min_{\mathbf{w},b} \; L = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \frac{C}{2}(\mathbf{1} - \mathbf{w}^\top\widehat{\mathbf{\Phi}} - b\mathbf{y})^\top\Lambda(\mathbf{1} - \mathbf{w}^\top\widehat{\mathbf{\Phi}} - b\mathbf{y}).$$

As this objective is convex, we can obtain the optimal solution of $\mathbf{w}, b$ by setting $\frac{\partial L}{\partial \mathbf{w}}$ and $\frac{\partial L}{\partial b}$ to zero:

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \Longrightarrow \mathbf{w} - C\widehat{\mathbf{\Phi}}\Lambda(\mathbf{1} - \widehat{\mathbf{\Phi}}^\top\mathbf{w} - b\mathbf{y}^\top) = \mathbf{0}.$$
$$\frac{\partial L}{\partial b} = 0 \Longrightarrow -\mathbf{y}\Lambda(\mathbf{1} - \widehat{\mathbf{\Phi}}^\top\mathbf{w} - b\mathbf{y}^\top) = 0.$$

By re-arranging the above equations, we can express them as a matrix equality,

$$\underbrace{\begin{bmatrix} \frac{\mathbf{I}}{C} + \widehat{\mathbf{\Phi}}\Lambda\widehat{\mathbf{\Phi}}^\top & \widehat{\mathbf{\Phi}}\Lambda\mathbf{y}^\top \\ \mathbf{y}\Lambda\widehat{\mathbf{\Phi}}^\top & \mathbf{y}\Lambda\mathbf{y}^\top \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} = \underbrace{\begin{bmatrix} \widehat{\mathbf{\Phi}}\Lambda\mathbf{1} \\ \mathbf{y}\Lambda\mathbf{1} \end{bmatrix}}_{\mathbf{z}}.$$

We absorb the coefficients on the left-hand side into a design matrix $\mathbf{M} \in \mathcal{R}^{d+1 \times d+1}$, and right-hand side into a vector $\mathbf{z} \in \mathcal{R}^{d+1}$. Consequently, we can express $\mathbf{w}$ and b as a function

of $\mathbf{M}^{-1}$ and $\mathbf{z}$, and derive their derivatives w.r.t. $[\phi]^s$ from the matrix inverse rule [92], leading to

$$\frac{\partial[\mathbf{w}^\top, b]^\top}{\partial[\phi_i]^s} = \mathbf{M}^{-1}\left(\frac{\partial \mathbf{z}}{\partial[\phi_i]^s} - \frac{\partial \mathbf{M}}{\partial[\phi_i]^s}\begin{bmatrix}\mathbf{w}\\ b\end{bmatrix}\right) \tag{2.31}$$

To compute the derivatives $\frac{\partial \mathbf{M}}{\partial[\phi]^s}$, we note that the upper left block of $\mathbf{M}$ is a $d \times d$ inner product matrix scaled by Λ and translated by $\frac{\mathbf{I}}{C}$, and we obtain the derivative w.r.t. each element (row r, column s) of the upper left block,

$$\frac{\partial(\frac{\mathbf{I}}{C} + \widehat{\mathbf{\Phi}}\Lambda\widehat{\mathbf{\Phi}}^\top)_{rs}}{\partial[\phi]^s(\mathbf{x}_i)} = \begin{cases}\mu_{y_i}[\phi]^r(\mathbf{x}_i) & \text{if } r \neq s,\\ 2\mu_{y_i}[\phi]^s(\mathbf{x}_i) & \text{if } r = s.\end{cases}$$

The remaining derivatives are $\frac{\partial \widehat{\mathbf{\Phi}}\Lambda\mathbf{y}^\top}{\partial[\phi]^s(\mathbf{x}_i)} = \mu_{y_i}$ and $\frac{\partial \mathbf{z}}{[\phi]^s(\mathbf{x}_i)} = [0, \ldots, y_i\mu_{y_i}, \ldots, 0]^\top \in \mathcal{R}^{d+1}$. To complete the chain rule in eq. (2.28), we also need

$$\frac{\partial \ell_{\mathcal{V}}}{\partial J} = -y_i\sigma(y_i J[\phi(\mathbf{x}_i)])(1 - \sigma(y_i J[\phi(\mathbf{x}_i)])). \tag{2.32}$$

Combining eqs. (2.29), (2.30), (2.31) and (2.32) completes the gradient $\frac{\partial \ell_{\mathcal{V}}}{\partial[\phi]^s}$.

Gradient w.r.t. C**.** The derivative $\frac{\partial J}{\partial C}$ is very similar to $\frac{\partial J}{\partial[\phi]^s}$, the difference being in $\frac{\partial \mathbf{M}}{\partial C}$, which only has non-zero value on diagonal elements,

$$\frac{\partial \mathbf{M}_{rs}}{\partial C} = \begin{cases}-\frac{1}{C^2} & \text{if } s = r \wedge r \neq m+1,\\ 0 & \text{otherwise.}\end{cases} \tag{2.33}$$

Although computing the derivative requires the inversion of matrix $\mathbf{M}$, $\mathbf{M}$ is only a $(d+1) \times (d+1)$ matrix. Because our algorithm converges after generating a few ($d \approx 100$) dimensions, the inverse operation is not computationally intensive.

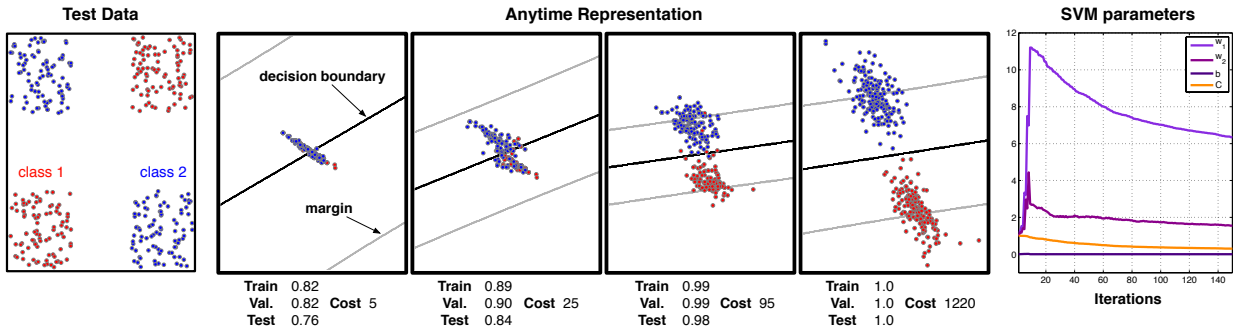

Figure 2.7: A demonstration of our method on a synthetic data set (shown at left). As the feature representation is allowed to use more expensive features, AFR can better distinguish the test data of the two classes. At the bottom of each representation is the classification accuracies of the training/validation/testing data and the cost of the representation. The rightmost plot shows the values of SVM parameters $\mathbf{w}, b$ and hyper-parameter C at each iteration.

2.2.4 Results

We evaluate our algorithm on a synthetic data set in order to demonstrate the AFR learning approach, as well as two benchmark data sets from very different domains: the Yahoo! Learning to Rank Challenge data set [18] and the Scene 15 recognition data set from [71].

Synthetic data. To visualize the learned anytime feature representation, we construct a synthetic data set as follows. We generate $n = 1000$ points (640 for training/validation and 360 for testing) uniformly sampled from four different regions of two-dimensional space (as shown in Figure 2.7, left). Each point is labeled to be in class 1 or class 2 according to the XOR rule. These points are then randomly-projected into a ten-dimensional feature space (not shown). Each of these ten features is assigned an extraction cost: {1, 1, 1, 2, 5, 15, 25, 70, 100, 1000}. Correspondingly, each feature f has zero-mean Gaussian noise added to it, with variance $\frac{1}{c_f}$ (where c_f is the cost of feature f). As such, cheap features are poorly representative of the classes while more expensive features more accurately distinguish the two classes. To highlight the feature-selection capabilities of our technique we set the evaluation cost c_e to 0. Using this data, we constrain the algorithm to learn a two-dimensional anytime representation (*i.e.* $\phi(\mathbf{x}) \in \mathcal{R}^2$).

The center portion of Figure 2.7 shows the anytime representations of testing points for various test-time budgets, as well as the learned hyperplane (black line), margins (gray lines) and classification accuracies. As the allowed feature cost budget is increased, AFR steadily adjusts the representation and classifier to better distinguish the two classes. Using a small set of features (cost = 95) AFR can achieve nearly perfect test accuracy and using all features AFR fully separates the test data.

The rightmost part of Figure 2.7 shows how the learned SVM classifier changes as the representation changes. The coefficients of the hyperplane $\mathbf{w} = [w_1, w_2]^\top$ initially change drastically to appropriately weight the AFR features, then decrease gradually as more weak learners are added to ϕ. Throughout, the hyper-parameter C is also optimized.

Yahoo Learning to Rank. The Yahoo! Learning to Rank Challenge data set consists of query-document instance pairs, with labels having values from $\{0, 1, 2, 3, 4\}$, where 4 means the document is perfectly relevant to the query and 0 means it is irrelevant. Following the steps of Chen et al. [24], we transform the data into a binary classification problem by distinguishing purely between relevant ($y_i \geq 3$) and irrelevant ($y_i < 3$) documents. The resulting labels are $y_i \in \{+1, -1\}$. The total binarized data set contains 2000, 2002, and 2001 training, validation and testing queries and 20258, 20258, 26256 query-document instances respectively. As in [24] we replicate each negative, irrelevant instance 10 times to simulate the scenario where only a few documents out of hundreds of thousands of candidate documents are highly relevant. Indeed in real world applications, the distribution of the two classes is often very skewed, with vastly more negative examples presented.

Each input contains 519 features, and the feature extraction costs are in the set {1, 5, 10, 20, 50, 100, 150, 200}. The unit of cost is the time required to evaluate one limited-depth regression tree $h_t(\cdot)$, thus the evaluation cost c_e is set to 1. To evaluate the cost-accuracy performance, we follow the typical convention for a binary ranking data set and use the Precision@5 metric. This counts how many documents are relevant in the top 5 retrieved documents for each query.

In order to address the label imbalance, we add a multiplicative weight to the loss of all positive examples, β_+, which is set by cross validation ($\beta_+ = 2$). We set the hyper-parameters to $T = 10$, $S = 20$ and $\lambda_0 = 10$. As the algorithm is by design fairly insensitive to hyper-parameters, this setting is determined without needing to search through (T, S, λ_0) space.

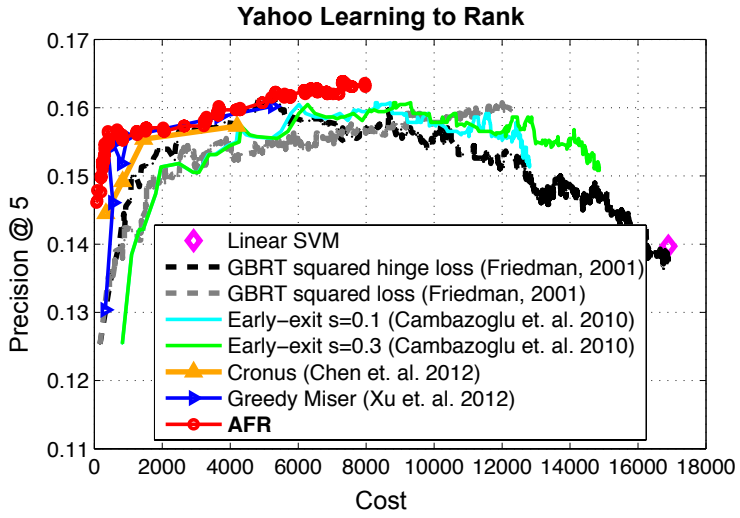

Figure 2.8: The accuracy/cost trade-off curves for a number of state-of-the-art algorithms on the Yahoo! Learning to Rank Challenge data set. The cost is measured in units of the time required to evaluate one weak learner.

Comparison. The most basic baseline is GBRT without cost consideration. We apply GBRT using two different loss functions: the squared loss and the un-regularized squared hinge loss. In total we train 2000 trees. We plot the cost and accuracy curves of GBRT by adding 10 trees at a time. In addition to this additive classifier, we show the results of a linear SVM applied to the original features as well.

We also compare against current state-of-the-art competing algorithms. We include *Early-Exit* [14], which is based on GBRT. It short-circuits the evaluation of lower ranked and unpromising documents at test-time, based on some threshold s (we show $s = 0.1, 0.3$), reducing the overall test-time cost. *Cronus* [24] improves over Early-Exit by reweighing and re-ordering the learned trees into a feature-cost sensitive cascade structure. We show results of a cascade with a maximum of 10 nodes. All of its hyper-parameters (cascade length, keep ratio, discount, early-stopping) are set based on the validation set. We generate the cost/accuracy curve by varying the trade-off parameter λ, in their paper. Finally, we compare against *Greedy Miser* described in Section 2.1 trained using the un-regularized squared hinge loss. The cost/accuracy curve is generated by re-training the algorithm with different cost/accuracy trade-off parameters λ. We also use the validation set to select the best number of trees needed for each λ.

Figure 2.8 shows the performance of all algorithms. Although the linear SVM uses all features to make cost-insensitive predictions, it achieves a relatively poor result on this ranking data set, due to the limited power of a linear decision boundary on the original feature space. This trend has previously been observed in [18]. GBRT with un-regularized squared hinge loss and squared loss achieve peak accuracy after using a significant amount of the feature set. Early-Exit only provides limited improvement over GBRT when the budget is low. This is primarily because, in this case, the test-time cost is dominated by feature extraction rather than the evaluation cost. Cronus improves over Early-Exit significantly due to its automatic stage reweighing and re-ordering. However, its power is still limited by its feature representation, which is not cost-sensitive. AFR out-performs the best performance of Greedy Miser for a variety of cost budgets. Different from Greedy Miser, which must be re-trained for different budgets along the cost/accuracy trade-off curve (each resulting in a different model), AFR consists of a single model which can be halted at any point along its curve—providing a state-of-the-art *anytime* classifier. It is noteworthy that AFR obtains the highest test-scores overall, which might be attributed to the better generalization of large-margin classifiers.

Scene recognition. The second data set we experiment with is from the image domain. The scene 15 [71] data set contains 4485 images from 15 scene classes. The task is to classify the scene in each image. Following the procedure used by Li et al. [76] and Lazebnik et al. [71], we construct the training set by selecting 100 images from each class, and leave the remaining 2865 images for testing. We extract a variety of vision features from Xiao et al. [130] with very different computational costs: GIST, spatial HOG, Local Binary Pattern (LBP), self-similarity, texton histogram, geometric texton, geometric color, and Object Bank [76]. As mentioned by the authors of Object Bank, each object detector works independently. Therefore we apply 177 object detectors to each image, and treat each of them as independent descriptors. In total, we have 184 different image descriptors, and the total number of resulting raw features is 76187. Note that we do not use the SVM features for this data set as we do in Section 2.1.4, because we would like to evaluate how well our algorithm performs on raw inputs. The feature extraction cost is the actual CPU time to compute each feature on a desktop with dual 6-core Intel i7 CPUs with 2.66GHz, ranging from 0.037s (Object Bank) to 9.282s (geometric texton). Since computing each type of image descriptor results in a group of features, as long as any of the features in a descriptor is requested, we extract the entire descriptor. Thus, subsequent requests for features in that descriptor are free.

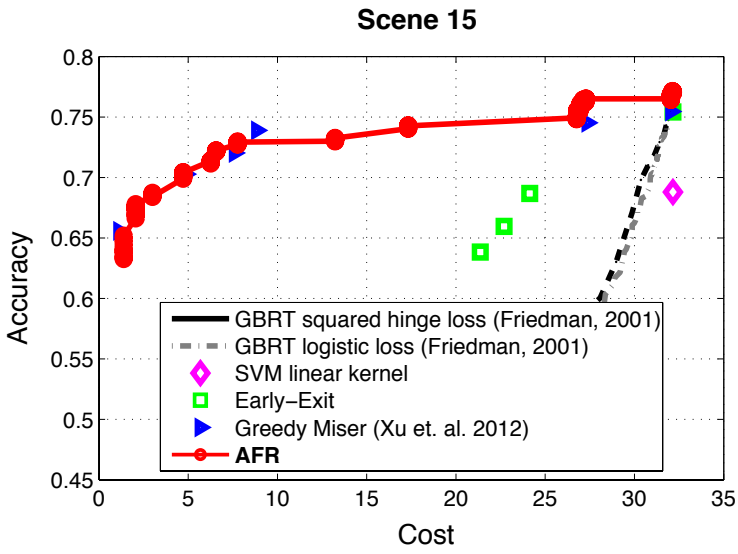

Figure 2.9: The accuracy/cost performance trade-off for different algorithms on the Scene 15 multi-class scene recognition problem. The cost is in units of CPU time.

We train 15 one-vs-all classifiers, and learn the feature representation mapping ϕ, the SVM parameters ($\mathbf{w}$,b,C) for each classifier separately. Since each descriptor is free once extracted, we also set the descriptor cost to zero whenever it is used by one of the 15 classifiers. To overcome the problem of different decision value scales resulting from different one-vs-all classifiers, we use the scaling method proposed by Platt [93] to re-scale each classifier prediction within $[0, 1]$.[4] We use the same hyper-parameters as the Yahoo! data set, except we set $\lambda_0 = 2^{10}$, as the unit of cost in scene15 is much smaller.

Figure 2.9 demonstrates the cost/accuracy performance of several current state-of-the-art techniques and our algorithm. The GBRT-based algorithms include GBRT using the logistic loss and the squared loss, where we use the scaling method described above for the hinge loss variant to cope with the scaling problem. We generate the curve by adding 10 trees at a time. Although these two methods achieve high accuracy, their costs are also significantly higher due to their cost-insensitive nature. We also evaluate a linear SVM. Because it is only able to learn a linear decision boundary on the *original* feature space, it has a lower accuracy than the GBRT-based techniques for a given cost. For cost-sensitive methods, we first evaluate

[4]The scaling method makes SVM predictions interpretable as probabilities. This can also be use to monitor the confidence threshold of the anytime classifiers to stop evaluation when a confidence threshold is met (*e.g.* in medical applications to avoid further costly feature extraction).

Early-Exit. As this is a multi-class classification problem, we introduce an early-exit every 10 trees, and we remove test inputs after scaling results in a score greater than a threshold s. We plot the curve by varying s. Since Early-Exit lacks the capability to automatically pick expensive and accurate features early-on, its improvement is very limited. For *Greedy Miser*, we split the training data into 75/25 and use the smaller subset as validation to set the number of trees. We use un-regularized squared hinge-loss with different values of the cost/accuracy trade-off parameter $\lambda \in \{4^0, 4^1, 4^2, 4^3, 4^4, 4^5\}$. Greedy Miser performs better than the previous baselines, and AFR consistently matches it, save one setting. Moreover, AFR generates a smoother budget curve, and can be stopped anytime to provide predictions at test-time.

2.2.5 Conclusion

To our knowledge, we provide the first learning algorithm for cost-sensitive anytime feature representations. Our results are highly encouraging, in particular Anytime Feature Representation Learning matches or even outperforms the results of the previously discussed classifiers learned under test-time resource constraints, which must be provided with knowledge about the exact test-time budget during training. Moreover, learning anytime *representations* adds new flexibility towards the choice of classifier and the learning setting and may enable new use cases and application areas.

Chapter 3

Classification with Trees and Cascades

In face detection applications, the majority of all image regions do not contain faces and can often be easily rejected based on the response of a few simple Haar features [125]. In some other applications, where data is not class-imbalanced like face detection, many inputs could be classified correctly based on a small subset of all available features, and this subset can vary across inputs. Applications like these two naturally inspire a different goal in controlling test-time cost and a different strategy: Controlling the *amortized* test-time cost and classification with trees and cascades.

3.1 Introduction

In the face detection example described above, different inputs require a variety of features to classify. Most inputs can be correctly classified by just Haar features, and only a few inputs need more complex vision features. Therefore, setting the same budget for every input will cause either a huge waste on easy-to-classify inputs or a too tight budget on hard-to-classify inputs. Instead, one should allocate different budgets for different inputs, giving hard-to-classify inputs more budget and giving easier ones less, and thus on-average, the test-time cost is within the budget. In other words, the *amortized* cost should be within the budget.

In this chapter, we propose a new algorithm, *Cost-Sensitive Tree of Classifiers (CSTC)* that aims to control the amortized test-time cost. CSTC minimizes an approximation of the exact expected test-time cost required to predict an instance. An illustration of a CSTC tree is shown in the right plot of Figure 3.1. As shown in the figure, because the input space is

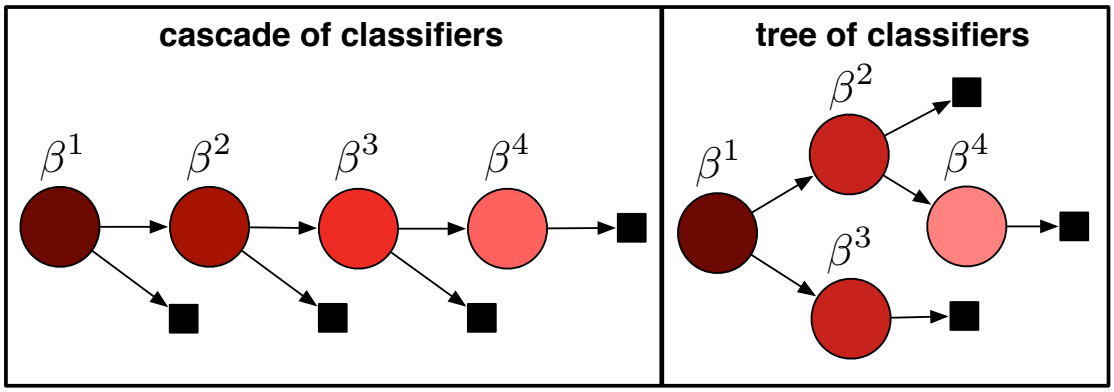

Figure 3.1: An illustration of two different techniques for learning under a test-time resource constraints. Circular nodes represent classifiers (with parameters $\boldsymbol{\beta}$) and black squares predictions. The color of a classifier node indicates the number of inputs passing through it (darker means more). *Left:* CSCC, a classifier cascade that optimizes the average cost by rejecting easier inputs early. *Right:* CSTC, a tree that trains expert leaf classifiers specialized on subsets of the input space.

partitioned by the tree, different features are only extracted where they are most beneficial. As a result, the amortized test-time cost is reduced. Unlike prior approaches, which reduce the total cost for every input [38] or which combine feature cost with mutual information to select features [34], a CSTC tree incorporates *input-dependent feature selection* into training and dynamically allocates higher feature budgets for infrequently traveled tree-paths. In data scenarios with highly skewed class imbalance, cascades might be a better model by rejecting many instances using a small number of features. We therefore apply the same test-time cost derivation to a stage-wise classifier for cascades. The resulting algorithm, *Cost-Sensitive Cascade of Classifiers (CSCC)*, is shown in the left plot of Figure 3.1.

3.2 Related Work

There is much previous work that addresses amortized test-time cost by building classifier cascades (mostly for binary classification) [125, 37, 74, 105, 96, 24, 100, 117]. They chain several classifiers into a sequence of stages. Each classifier can either "early-exit" inputs (making a final prediction), or pass them on to the next stage. This decision is made based on the prediction of each instance. Different from CSCC, these algorithms typically do not

take into account feature cost and implement more ad-hoc rules to trade-off accuracy and cost.

For learning tasks with *balanced classes* and *specialized features*, the linear cascade model is less well-suited. Because all inputs follow exactly the same linear path, it cannot capture the scenario in which different subsets of inputs require different expert features. Chai et al. [15] introduce the value of un-extracted features, where the value of a feature is the increase (gain) in expected classification accuracy minus the cost of including that feature. During test-time, at each iteration, their algorithm picks the feature that has the highest value and retrains a classifier with the new feature. The algorithm stops when there is no increase in expected accuracy, or all features are included. Because they employ a naive Bayes classifier, retraining incurs very little cost. Similarly, Gao and Koller [45] use locally weighted regression during test-time to predict the information gain of unknown features.

Most recently, Karayev et al. [61] use reinforcement learning during test-time to dynamically select object detectors for a particular image. Our approach shares the same idea that different inputs require different features. However, instead of learning the best feature for each input during test-time, which introduces an additional cost, we learn and fix a tree structure in training. Each branch of the tree only handles a subset of the input space and, as such, the classifiers in a given branch become specialized for those inputs. Moreover, because we learn a fixed tree structure, it has a test-time complexity that is constant with respect to the training set size.

Concurrently, there has been work proposed by Deng et al. [33] to speed up the training and evaluation of tree-structured classifiers (specifically label trees [4]), by avoiding many binary one-vs-all classifier evaluations. However, in many real world datasets the test-time cost is largely composed of feature extraction time and our aim is different from their work. Possibly most similar to our work is by Busa-Fekete et al. [13], who apply a Markov decision process to learn a directed acyclic graph. At each step, they select features for different instances. Although similar in motivation, their algorithmic framework is very different and can be regarded as complementary to ours.

It is worth mentioning that, although Hierarchical Mixture of Experts (HME) [60] also builds tree-structured classifiers, it does not consider reducing the test-time cost and thus results in fundamentally different models. In contrast, we train each classifier with the test-time cost

in mind and each test input only traverses a *single* path from the root down to a terminal element, accumulating path-specific costs. In HME, all test inputs traverse all paths and all leaf-classifiers contribute to the final prediction, incurring the same cost for all test inputs.

3.3 Background

We learn a classifier $H : \mathcal{R}^d \rightarrow \mathcal{Y}$, parameterized by $\boldsymbol{\beta}$, to minimize a continuous, non-negative loss function ℓ,

$$\min_{\boldsymbol{\beta}} \frac{1}{n} \sum_{i=1}^{n} \ell(H(\mathbf{x}_i; \boldsymbol{\beta}), y_i).$$

We assume that H is a linear classifier, $H(\mathbf{x}; \boldsymbol{\beta}) = \mathbf{x}^\top \boldsymbol{\beta}$. To avoid overfitting, we deploy a standard l_1 regularization term, $|\boldsymbol{\beta}|$ to control model complexity. This regularization term has the known side-effect of keeping $\boldsymbol{\beta}$ sparse ([116]), which requires us to only evaluate a subset of all features. In addition, to balance the test-time cost incurred by the classifier, we also incorporate the cost term $c(\boldsymbol{\beta})$ described in the following section. The combined test-time cost-sensitive optimization becomes

$$\min_{\boldsymbol{\beta}} \underbrace{\frac{1}{n} \sum_{i} \ell(\mathbf{x}_i^\top \boldsymbol{\beta}, y_i) + \rho|\boldsymbol{\beta}|}_{\text{regularized loss}} \; + \; \lambda \underbrace{c(\boldsymbol{\beta})}_{\text{test-cost}}, \tag{3.1}$$

where λ is the accuracy/cost trade-off parameter, and ρ controls the strength of the regularization.

The test-time cost of H is regulated by the features extracted for that classifier. We denote the extraction cost for feature α as c_α. The cost $c_\alpha \geq 0$ is suffered at most once, only for the initial extraction, as feature values can be cached for future use. For a classifier H, parameterized by $\boldsymbol{\beta}$, we can record the features used:

$$\|\beta_\alpha\|_0 = \begin{cases} 1 & \text{if feature } \alpha \text{ is used in } H \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Here, $\|\cdot\|_0$ denotes the l_0 norm with $\|x\|_0 = 1$ if $x \neq 0$ and $\|x\|_0 = 0$ otherwise. With this notation, we can formulate the total test-time cost required to evaluate a test input $\mathbf{x}_i$ with classifier H (and parameters $\boldsymbol{\beta}$) as

$$\sum_{\alpha=1}^{d} c_\alpha \|\beta_\alpha\|_0. \tag{3.3}$$

Note that for now we assume all inputs have the same feature extraction cost, and the amortized test-time cost equals to the cost of each input,

$$c(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n} 1 \sum_{\alpha=1}^{d} c_\alpha \|\beta_\alpha\|_0 = \sum_{\alpha=1}^{d} c_\alpha \|\beta_\alpha\|_0. \tag{3.4}$$

3.4 Cost-sensitive tree of classifiers

We introduce an algorithm that aims to reduce amortized test-time cost by giving different inputs different budgets. Our algorithm employs a tree structure to extract particular features for particular inputs, and we refer to it as the *Cost-Sensitive Tree of Classifiers* (*CSTC*). We begin by introducing foundational concepts regarding the CSTC tree and derive a global cost term that extends eq. (3.4) to trees of classifiers. Then we relax the resulting loss function into a well-behaved optimization problem.

CSTC nodes. The fundamental building block of the CSTC tree is a CSTC node—a linear classifier as described in Section 3.3. Our classifier design is based on the assumption that instances with similar labels tend to have similar relevant features. Thus, we design our tree algorithm to partition the input space based on classifier predictions. Intermediate classifiers determine the path of instances through the tree and leaf classifiers become experts for a small subset of the input space.

Correspondingly, there are two different nodes in a CSTC tree (depicted in Figure 3.2): *classifier nodes* (white circles) and *terminal elements* (black squares). Each *classifier node* v^k is associated with a weight vector $\boldsymbol{\beta}^k$ and a threshold θ^k. These classifier nodes branch inputs by their threshold θ^k, sending inputs to their upper child if $\mathbf{x}_i^\top \boldsymbol{\beta}^k > \theta^k$, and to their lower child otherwise. *Terminal elements* are "dummy" structures and are *not* real classifiers.

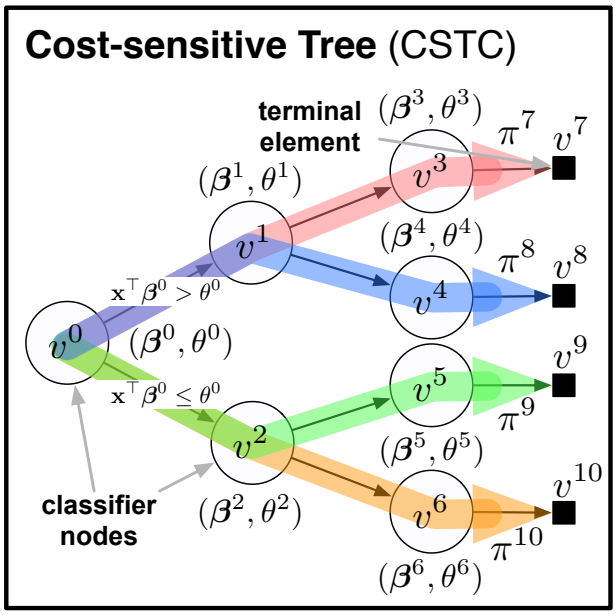

Figure 3.2: A schematic layout of a CSTC tree. Each node v^k is associated with a weight vector $\boldsymbol{\beta}^k$ for prediction and a threshold θ^k to send instances to different parts of the tree. We solve for $\boldsymbol{\beta}^k$ and θ^k that best balance the accuracy/cost trade-off for the whole tree. Each path in the CSTC tree is shown in a different color.

They return the predictions of their direct parent classifier nodes—essentially functioning as a placeholder for an exit out of the tree. The tree structure may be a full balanced binary tree of some depth (*e.g.* Figure 3.2), or can be pruned based on a validation set. For simplicity, we assume at this point that nodes with terminal element children must be leaf nodes (as depicted in the figure)—an assumption that we will relax later on.

During test-time, inputs traverse through the tree, starting from the root node v^0. The root node produces predictions $\mathbf{x}_i^\top \boldsymbol{\beta}^0$ and sends the input $\mathbf{x}_i$ along one of two different paths, depending on whether $\mathbf{x}_i^\top \boldsymbol{\beta}^0 > \theta^0$. By repeatedly branching the test inputs, classifier nodes sitting deeper in the tree only handle a small subset of all inputs and become specialized towards that subset of the input space.

3.4.1 CSTC Loss

In this subsection, we discuss the loss and test-time cost of a CSTC tree. We then derive a single global loss function over all nodes in the CSTC tree.

Soft tree traversal. As we described before, inputs are partitioned at each node during test-time, and we use a hard threshold to achieve this partitioning. However, modeling a CSTC tree with hard thresholds leads to a combinatorial optimization problem that is NP-hard [68, Chapter 15]. As a remedy, during training, we *softly* partition the inputs and assign *traversal probabilities* $p(v^k|\mathbf{x}_i)$ to denote the likelihood of input $\mathbf{x}_i$ traversing through node v^k. Every input $\mathbf{x}_i$ traverses through the root, so we define $p(v^0|\mathbf{x}_i)=1$ for all i. We define a "sigmoidal" soft belief that an input $\mathbf{x}_i$ will transition from classifier node v^k with threshold θ^k to its *upper* child v^u as

$$p(v^u|\mathbf{x}_i, v^k) = \frac{1}{1+\exp(-(\mathbf{x}_i^\top \boldsymbol{\beta}^k - \theta^k))}. \tag{3.5}$$

Let v^k be a node with upper child v^u and lower child v^l. We can express the probabilities of reaching nodes v^u and v^l recursively as $p(v^u|\mathbf{x}_i) = p(v^u|\mathbf{x}_i, v^k)p(v^k|\mathbf{x}_i)$ and $p(v^l|\mathbf{x}_i) = \big[1-p(v^u|\mathbf{x}_i, v^k)\big]p(v^k|\mathbf{x}_i)$ respectively. Note that it follows immediately, that if $\mathcal{V}^d$ contains all nodes at tree-depth d, we have

$$\sum_{v\in\mathcal{V}^d} p(v|\mathbf{x}) = 1. \tag{3.6}$$

In the following paragraphs we incorporate this probabilistic framework into the loss and cost terms of (3.1) to obtain the corresponding *expected tree loss* and *expected tree cost*.

Expected tree loss. To obtain the *expected tree loss*, we sum over all nodes $\mathcal{V}$ in a CSTC tree and all inputs and weigh the loss $\ell(\cdot)$ of input $\mathbf{x}_i$ at node v^k by the probability that the input reaches v^k, $p_i^k = p(v^k|\mathbf{x}_i)$,

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{v^k\in\mathcal{V}} p_i^k \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i). \tag{3.7}$$

This has two effects: 1. the local loss for each node focuses more on likely inputs; 2. the global objective attributes more weight to classifiers that serve many inputs. Technically, the prediction of the CSTC tree is made entirely by the terminal nodes (*i.e.* the leaves), and an obvious suggestion may be to only minimize their classification loss and leave the interior nodes as "gates" without any predictive abilities. However, such a setup creates local minima that send all inputs to the terminal node with the lowest initial error rate. These local minima are hard to escape and therefore we found it to be important to minimize the loss for *all* nodes. Effectively, this forces a structure onto the tree that similarly labeled inputs leave through similar leaves and achieves robustness by assigning high loss to such pathological solutions.

Expected tree costs. The cost of a test input is the cumulative cost across all classifiers along its path through the CSTC tree. Figure 3.2 illustrates an example of a CSTC tree with all paths highlighted in different colors. Every test input must follow along exactly one of the paths from the root to a terminal element. Let $\mathcal{L}$ denote the set of all terminal elements (*e.g.*, in Figure 3.2 we have $\mathcal{L}=\{v^7, v^8, v^9, v^{10}\}$), and for any $v^l \in \mathcal{L}$, let π^l denote the set of all *classifier nodes* along the unique path from the root v^0 before terminal element v^l (*e.g.*, $\pi^9=\{v^0, v^2, v^5\}$).

For an input $\mathbf{x}_i$, exiting through terminal node v^l, a feature α needs to be extracted if and only if at least one classifier along the path π^l uses this feature. We extend the indicator function defined in (3.2) accordingly:

$$\left\| \sum_{v^j \in \pi^l} \left|\beta_\alpha^j\right| \right\|_0 = \begin{cases} 1 & \text{if feature } \alpha \text{ is used along path to terminal node } v^l \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

We can extend the cost term in eq. (3.4) to capture the traversal cost from root to node v^l as

$$c^l = \sum_\alpha c_\alpha \left\| \sum_{v^j \in \pi^l} |\beta_\alpha^j| \right\|_0 . \tag{3.9}$$

Given an input $\mathbf{x}_i$, the expected cost is then $E[c^l|\mathbf{x}_i] = \sum_{l \in \mathcal{L}} p(v^l|\mathbf{x}_i) c^l$. To approximate the data distribution, we sample uniformly at random from our training set, *i.e.* we set

$p(\mathbf{x}_i) \approx \frac{1}{n}$, and obtain the unconditional expected cost

$$E[cost] = \sum_{i=1}^{n} p(\mathbf{x}_i) \sum_{l \in \mathcal{L}} p(v^l|\mathbf{x}_i) c^l \approx \sum_{l \in \mathcal{L}} c^l \underbrace{\sum_{i=1}^{n} p(v^l|\mathbf{x}_i) \frac{1}{n}}_{:=p^l} = \sum_{l \in \mathcal{L}} c^l p^l. \tag{3.10}$$

Here, p^l denotes the probability that a randomly picked training input exits the CSTC tree through terminal node v^l. We can combine (3.9), (3.10) with (3.7) and obtain the objective function,

$$\sum_{v^k \in \mathcal{V}} \underbrace{\left(\frac{1}{n} \sum_{i=1}^{n} p_i^k \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \sum_{v^l \in \mathcal{L}} p^l \underbrace{\left[\sum_{\alpha} c_\alpha \left\| \sum_{v^j \in \pi^l} |\beta_\alpha^j| \right\|_0 \right]}_{\text{test-time cost}}, \tag{3.11}$$

where we use the abbreviations $p_i^k = p(v^k|\mathbf{x}_i)$ and $\ell_i^k = \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$.

3.4.2 Test-cost Relaxation

The cost penalties in (3.11) are exact but difficult to optimize due to the discontinuity and non-differentiability of the l_0 norm. As a solution, throughout this section we use the mixed-norm relaxation of the l_0 norm over sums,

$$\sum_j \left\| \sum_i |A_{ij}| \right\|_0 \rightarrow \sum_j \sqrt{\sum_i (A_{ij})^2}, \tag{3.12}$$

described by Kowalski [69]. Note that for a vector, this relaxes the l_0 norm to the l_1 norm, *i.e.* $\sum_j \|a_j\|_0 \rightarrow \sum_j \sqrt{(a_j)^2} = \sum_j |a_j|$, recovering the commonly used approximation to encourage sparsity. For matrices $\mathbf{A}$, the mixed norm applies the l_1 norm over rows and the l_2 norm over columns, thus encouraging a whole row to be all-zero or non-sparse. In our case this has the natural interpretation to encourage re-use of features that are already extracted along a path. Using the relaxation in (3.12) on the l_0 norm in (3.11) gives the final optimization

problem:

$$\min_{\boldsymbol{\beta}^0,\theta^0,\ldots,\boldsymbol{\beta}^{|V|},\theta^{|V|}} \underbrace{\sum_{v^k\in\mathcal{V}}\left(\frac{1}{n}\sum_{i=1}^{n} p_i^k \ell_i^k + \rho|\boldsymbol{\beta}^k|\right)}_{\text{regularized loss}} + \lambda \underbrace{\sum_{v^l\in\mathcal{L}} p^l \left[\sum_{\alpha} c_\alpha \sqrt{\sum_{v^j\in\pi^l}(\beta_\alpha^j)^2}\right]}_{\text{test-time cost penalty}} \tag{3.13}$$

We can illustrate the fact that the mixed-norm encourages re-use of features with a simple example. If two classifiers $v^k \neq v^{k'}$ along a path π^l use *different* features with identical weight, *i.e.* $\beta_t^k = \epsilon = \beta_s^{k'}$ and $t \neq s$, the test-time cost penalty along π^l is $\sqrt{\epsilon^2} + \sqrt{\epsilon^2} = 2\epsilon$. However, if the two classifiers *re-use* the same feature, *i.e.* $t = s$, the test-time cost penalty reduces to $\sqrt{\epsilon^2 + \epsilon^2} = \sqrt{2}\epsilon$.

3.4.3 Optimization

There are many techniques to minimize the objective function (3.13). We use block coordinate descent, optimizing with respect to the parameters of a single classifier node v^k at a time, keeping all other parameters fixed. To minimize (3.13) (up to a local minimum) with respect to parameters $\boldsymbol{\beta}^k, \theta^k$ we use the lemma below to overcome the non-differentiability of the square-root term (and l_1 norm, which we can rewrite as $|a| = \sqrt{a^2}$) resulting from the l_0-relaxation (3.12).

Lemma 1. *Given a positive function $g(x)$, the following holds:*

$$\sqrt{g(x)} = \inf_{z>0} \frac{1}{2}\left[\frac{g(x)}{z} + z\right]. \tag{3.14}$$

It is straight-forward to see that $z = \sqrt{g(x)}$ minimizes the function on the right hand side and satisfies the equality, which leads to the proof of the lemma.

For each square-root or l_1 term we 1) introduce an auxiliary variable (*i.e.* z above), 2) substitute in (3.14), and 3) alternate between minimizing the objective in (3.13) with respect to $\boldsymbol{\beta}^k, \theta^k$ and solving for the auxiliary variables. The former minimization is performed with conjugate gradient descent and the latter can be computed efficiently in closed form. This pattern of block-coordinate descent followed by a closed form minimization is repeated until

convergence. Note that the objective is guaranteed to converge to a fixed point because each iteration decreases the objective function, which is bounded below by zero. In the following subsection, we detail the block coordinate descent optimization technique. Lemma 1 is only defined for strictly positive functions $g(x)$. As we are performing function minimization, we can reach cases where $g(x) = 0$ and Lemma 1 is ill defined. Thus, as a practical work-around, we clamp values to zero once they are below a small threshold (10^{-4}).

Optimization details. As terminal nodes are only placeholders and do not have their own parameters, we only focus on classifier nodes, which are depicted as round circles in Figure 3.2.

Leaf nodes. The optimization of leaf nodes (*e.g.* v^3, v^4, v^5, v^6 in Figure 3.2) is simpler because there are no downstream dependencies. Let v^k be such a classifier node with only a single "dummy" terminal node $v^{k'}$. During optimization of (3.13), we fix all other parameters $\boldsymbol{\beta}^j, \theta^j$ of other nodes v^j and the respective terms become constants. Therefore, we remove all other paths, and only minimize over the path $\pi^{k'}$ from the root to terminal node $v^{k'}$. Even along the path $\pi^{k'}$ most terms become constant and the only non-constant parameter is $\boldsymbol{\beta}^k$ (the branching parameter θ^k can be set to $-\infty$ because v^k has only one child). We color non-constant terms in the remaining function in blue below,

$$\frac{1}{n}\sum_{i=1}^{n} p_i^k \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i) + \rho|\boldsymbol{\beta}^k| + \lambda\, p^{k'} \left[\sum_\alpha c_\alpha \sqrt{(\beta_\alpha^k)^2 + \sum_{v^j \in \pi^{k'} \setminus v^k} (\beta_\alpha^j)^2} \right], \tag{3.15}$$

where the notation $\mathcal{S} \setminus b$ denotes a set containing all of the elements in $\mathcal{S}$ except b. After identifying the non-constant terms, we can apply Lemma 1, making (3.15) differentiable with respect to β_α^k. Let us define auxiliary variables γ_α and η_α for $1 \le \alpha \le d$ for the l_1-regularization term and the test-time cost term. Further, let us collect the constants in the test-time cost term $s_{\text{test-time}} = \sum_{v^j \in \pi^{k'} \setminus v^k} (\beta_\alpha^j)^2$. Applying Lemma 1 results in the following substitutions:

$$\begin{aligned}
&\sum_\alpha \rho|\beta_\alpha^k| = \sum_\alpha \rho\sqrt{(\beta_\alpha^k)^2} \longrightarrow \sum_\alpha \rho \frac{1}{2}\left(\frac{(\beta_\alpha^k)^2}{\gamma_\alpha} + \gamma_\alpha \right), \\
&\sum_\alpha c_\alpha \sqrt{(\beta_\alpha^k)^2 + s_{\text{test-time}}} \longrightarrow \sum_\alpha c_\alpha \frac{1}{2}\left(\frac{(\beta_\alpha^k)^2 + s_{\text{test-time}}}{\eta_\alpha} + \eta_\alpha \right).
\end{aligned} \tag{3.16}$$

As a result, we obtain a differentiable objective function after making the above substitutions. We can solve $\boldsymbol{\beta}^k$ by alternately minimizing the obtained differentiable function w.r.t. $\boldsymbol{\beta}^k$ with $\gamma_\alpha, \eta_\alpha$ fixed, and minimizing $\gamma_\alpha, \eta_\alpha$ with $\boldsymbol{\beta}^k$ fixed (*i.e.* minimizing η_α is equivalent to setting $\eta_\alpha = \sqrt{(\beta_\alpha^k)^2 + s_{\text{test-time}}}$). Recall that θ^k does not require optimization as v^k does not further branch inputs.

It is straight-forward to show [6, page 72], that the right hand side of Lemma 1 is jointly convex in x and z, so as long as $g(x)$ is a quadratic function of x. Thus, if $\ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$ is the squared loss, the substituted objective function is jointly convex in $\boldsymbol{\beta}^k$ and in $\gamma_\alpha, \eta_\alpha$, and therefore we can obtain a globally-optimal solution. Moreover, we can solve $\boldsymbol{\beta}^k$ in closed form. Let us define three design matrices

$$\mathbf{X}_{i\alpha} = [\mathbf{x}_i]_\alpha, \quad \boldsymbol{\Omega}_{ii} = p_i^k, \quad \boldsymbol{\Gamma}_{\alpha\alpha} = \frac{\rho}{\gamma_\alpha} + \lambda\Big(\frac{p^k c_\alpha}{\eta_\alpha}\Big),$$

where $\boldsymbol{\Omega}$ and $\boldsymbol{\Gamma}$ are both diagonal and $[\mathbf{x}_i]_\alpha$ is the α feature of instance $\mathbf{x}_i$. The closed-form solution for $\boldsymbol{\beta}^k$ is as follows,

$$\boldsymbol{\beta}^k = (\mathbf{X}^\top \boldsymbol{\Omega} \mathbf{X} + \boldsymbol{\Gamma})^{-1} \mathbf{X}^\top \boldsymbol{\Omega} \mathbf{y}. \tag{3.17}$$

Intermediate nodes. We further generalize this approach to all classifier nodes. As before, we optimize one node at a time, fixing the parameters of all other nodes. However, optimizing the parameters $\boldsymbol{\beta}^k, \theta^k$ of an *internal* node v^k, which has two children affects the parameters of descendant nodes. This affects the optimization of the regularized classifier loss and the test-time cost separately. We state how these terms in the global objective (3.13) are affected, and then show how to minimize it.

Let $\mathcal{S}$ be the set containing all descendant nodes of v^k. Changes to the parameters $\boldsymbol{\beta}^k, \theta^k$ will affect the traversal probabilities p_i^j for all $v^j \in \mathcal{S}$ and therefore enter the downstream loss functions. We first state the regularized loss part of (3.13) and once again color non-constant parameters in blue,

$$\frac{1}{n} \sum_i p_i^k \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i) + \frac{1}{n} \sum_{v^j \in \mathcal{S}} \sum_i p_i^j \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^j, y_i) + \rho |\boldsymbol{\beta}^k|. \tag{3.18}$$

Algorithm 5 CSTC global optimization

Input: data $\{\mathbf{x}_i, y_i\} \in \mathcal{R}^d \times \mathcal{R}$, initialized CSTC tree
repeat
  for $k = 1$ **to** $N = \#$ CSTC nodes **do**
    repeat
      Solve for γ, η (fix $\boldsymbol{\beta}^k, \theta^k$) using left hand side of (3.16)
      Solve for $\boldsymbol{\beta}^k, \theta^k$ (fix γ, η) with conjugate gradient descent, or in closed-form
    until objective changes less than ε
  end for
until objective changes less than ϵ

For the cost terms in (3.13), recall that the cost of each path π^l is weighted by the probability p^l of traversing that path. Changes to $\boldsymbol{\beta}^k, \theta^k$ affect the probability of any path that passes through v^k and its corresponding probability p^l. Let $\mathcal{P}$ be the terminal elements associated with paths passing through v^k. We state the cost function with non-constant parameters in blue,

$$\sum_{v^l \in \mathcal{P}} p^l \underbrace{\left[\sum_{\alpha} c_\alpha \sqrt{\left(\sum_{v^j \in \pi^l \setminus v^k} (\beta_\alpha^j)^2 + (\beta_\alpha^k)^2 \right)} \right]}_{\text{test-time cost}} \tag{3.19}$$

Adding (3.18) and (3.19), with the latter weighted by λ, gives the internal node loss. To make the combined objective function differentiable we apply Lemma 1 to the l_1-regularization, and test-time cost terms and introduce auxiliary variables $\gamma_\alpha, \eta_\alpha$ as in (3.16). Similar to the leaf node case, we solve $\boldsymbol{\beta}^k, \theta^k$ by alternately minimizing the new objective w.r.t. $\boldsymbol{\beta}^k, \theta^k$ with $\gamma_\alpha, \eta_\alpha$ fixed, and minimizing $\gamma_\alpha, \eta_\alpha$ with fixed $\boldsymbol{\beta}^k, \theta^k$. Unlike leaf nodes, optimizing the objective function w.r.t. $\boldsymbol{\beta}^k, \theta^k$ cannot be expressed in closed form even with squared loss. Therefore, we optimize it with conjugate gradient descent. Algorithm 5 describes how the entire CSTC tree is optimized.

Node initialization. The minimization of (3.13) is non-convex and is therefore initialization dependent. However, minimizing (3.13) with respect to the parameters of leaf classifiers *is convex*. We therefore initialize the tree top-to-bottom, starting at v^0, and optimizing over $\boldsymbol{\beta}^k$ by minimizing (3.13) while considering all descendant nodes of v^k as "cut-off" (thus

pretending node v^k is a leaf). This initialization is also very fast in the case of a quadratic loss, as it can be solved for in closed form.

3.4.4 Fine-tuning

The original test-time cost term in (3.4) sums over the cost of all features that are extracted during test-time. The relaxation in eq. (3.12) makes the exact l_0 cost differentiable and is still well suited to select which features to extract. However, the mixed-norm does also impact the performance of the classifiers, because (different from the l_0 norm) larger weights in $\boldsymbol{\beta}$ incur larger cost penalties. We therefore introduce a post-processing step to correct the classifiers from this unwanted regularization effect. We re-optimize the loss of all *leaf* classifiers (*i.e.* classifiers that make final predictions), while clamping all features with zero-weight to strictly remain zero.

$$\begin{aligned} &\min_{\bar{\boldsymbol{\beta}}^k} \sum_i p_i^k \ell(\mathbf{x}_i^\top \bar{\boldsymbol{\beta}}^k, y_i) + \rho|\bar{\boldsymbol{\beta}}^k| \\ &\text{s.t. } \bar{\beta}_t^k = 0 \text{ if } \beta_t^k = 0. \end{aligned}$$

Here, we do not include the cost-term, because the decision regarding which features to use is already made. The final CSTC tree uses these re-optimized weight vectors $\bar{\boldsymbol{\beta}}^k$ for all leaf classifier nodes v^k.

3.4.5 Determining the tree structure

As the CSTC tree does not need to be balanced, its structure is an implicit parameter of the algorithm. We discuss a approach to determine the structure of the tree in the absence of prior knowledge. We build a full (balanced) CSTC tree of depth d and initialize all nodes. To obtain a more compact model and to avoid over-fitting, the CSTC tree can be pruned with the help of a validation set. We compute the validation error of the initialized CSTC tree at each node. Starting with the leaf nodes, we then prune away nodes that, upon removal, do not decrease the validation performance (in the case of ranking data, we even can use

validation NDCG as our pruning criterion). After pruning, the tree structure is fixed and all nodes are optimized with the procedure described above.

3.5 Cost-sensitive Cascade of Classifiers

Many real world applications have data distributions with high class imbalance. One example is face detection, where the vast majority of all image patches do not contain faces; another example is web-search ranking, where almost all web-pages are irrelevant to a given query. Often, a few features may suffice to detect that an image does not contain a face or that a web-page is irrelevant. Further, in applications such as web-search ranking, the accuracy on low relevance instances is less important as long as they are not mistakenly predicted to be highly relevant (and therefore are not displayed to the end user).

In these settings, the entire focus of the algorithm should be on the most confident positive samples. Sub-trees that lead to only negative predictions, can be pruned effectively as there is no value in providing fine-grained differentiation between negative samples. This further reduces the average feature cost, as negative inputs traverse through shorter paths and require fewer features to be extracted. Previous work obtains these unbalanced trees by explicitly learning cascade structured classifiers [125, 37, 74, 105, 24, 117]. CSTC can incorporate cascades naturally as a special case, in which the tree of classifiers has only a single node per level of depth. However, further modifications can be made to accommodate the specifics of these settings. We introduce two changes to the learning algorithm:

- Inputs of different classes are re-weighted to account for the severe class imbalance.
- Every classifier node v^k has a terminal element as child and is weighted by the probability of *exiting* rather than the probability of reaching node v^k.

We refer to the modified algorithm as *Cost-Sensitive Cascade of Classifiers (CSCC)*. An example cascade is illustrated in Figure 3.3. A CSCC with K-stages is defined by a set of weight vectors $\boldsymbol{\beta}^k$ and thresholds θ^k, $\mathcal{C} = \{(\boldsymbol{\beta}^1, \theta^1), (\boldsymbol{\beta}^2, \theta^2), \cdots, (\boldsymbol{\beta}^K, -)\}$. An input is early-exited from the cascade at node v^k if $\mathbf{x}^\top \boldsymbol{\beta}^k < \theta^k$ and is sent to its terminal element v^{k+1}.

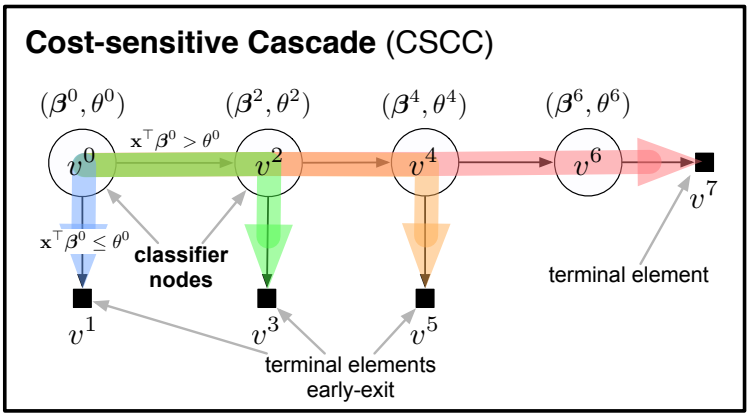

Figure 3.3: Schematic layout of our classifier cascade with four classifier nodes. All paths are colored in different colors.

Otherwise, the input is sent to the next classifier node. At the final node v^K a prediction is made for all remaining inputs via $\mathbf{x}^\top \boldsymbol{\beta}^K$.

In CSTC, most classifier nodes are internal and branch inputs. As such, the predictions need to be similarly accurate for all inputs to ensure that they are passed on to the correct part of the tree. In CSCC, each classifier node early-exits a fraction of its inputs, providing their *final* prediction. As mistakes of such exiting inputs are irreversible, the classifier needs to ensure particularly low error rates for this fraction of inputs. All other inputs are passed down the chain to later nodes. This key insight inspires us to modify the loss function of CSCC from the original CSTC formulation in (3.7). Instead of weighting the contribution of classifier loss $\ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$ by p_i^k, the probability of input $\mathbf{x}_i$ *reaching* node v^k, we weight it with p_i^{k+1}, the probability of *exiting* through terminal node v^{k+1}. As a second modification, we introduce an optional class-weight $\mu_{y_i} > 0$ which absorbs some of the impact of the class imbalance. The resulting loss becomes:

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{v^k \in \mathcal{V}} \mu_{y_i} p_i^{k+1} \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i). \tag{3.20}$$

The cost term is unchanged and the combined cost-sensitive loss function of CSCC becomes

$$\underbrace{\sum_{v^k \in \mathcal{V}} \left(\frac{1}{n} \sum_{i=1}^{n} \mu_{y_i} p_i^{k+1} \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \underbrace{\sum_{v^l \in L} p^l \left[\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} (\beta_\alpha^j)^2} \right]}_{\text{feature cost penalty}} . \tag{3.21}$$

We optimize (3.21) using the same block coordinate descent optimization described in Section 3.4.3. As before, we initialize the cascade from left to right, while assuming the currently initialized node is the last node.

3.6 Extension to non-linear classifiers

Although CSTC's decision boundary may be non-linear, each individual node classifier is linear. For many problems this may be too restrictive and insufficient to divide the input space effectively. In order to allow non-linear decision boundaries we map the input into a more expressive feature space with the "boosting trick" [44, 23], prior to our optimization. In particular, we first train gradient boosted regression trees with a squared loss penalty for T iterations and obtain a classifier $H'(\mathbf{x}_i) = \sum_{t=1}^{T} h_t(\mathbf{x}_i)$, where each function $h_t(\cdot)$ is a limited-depth CART tree [7]. We then define the mapping $\mathbf{h}(\mathbf{x}_i) = [h_1(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)]^\top$ and apply it to all inputs. The boosting trick is particularly well suited for our feature cost sensitive setting, as each CART tree only uses a small number of features. Nevertheless, this pre-processing step does affect the loss function in two ways: 1. the feature extraction now happens within the CART trees; and 2. the evaluation time of the CART trees needs to be taken into account.

Feature cost after the boosting trick. After the transformation $\mathbf{x}_i \rightarrow \mathbf{h}(\mathbf{x}_i)$, each input is $T-$dimensional and consequently, we have the weight vectors $\boldsymbol{\beta} \in \mathcal{R}^T$. To incorporate the feature extraction cost into our loss, we define an auxiliary matrix $\mathbf{F} \in \{0,1\}^{d \times T}$ with $F_{\alpha t} = 1$ if and only if the CART tree h_t uses feature α. With this notation, we can incorporate the CART-trees into the original feature extraction cost term for a weight vector $\boldsymbol{\beta}$, as stated in (3.4). The new formulation and its relaxed version (following the mixed-norm relaxation as

stated in (3.12)) are then:

$$\sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t=1}^{T} |F_{\alpha t}\beta_t| \right\|_0 \longrightarrow \sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{t=1}^{T} (F_{\alpha t}\beta_t)^2}.$$

The non-negative sum inside the l_0 norm is non-zero if and only if feature α is used by at least one tree with non-zero weight, *i.e.* $|\beta_t| > 0$. Similar to a single classifier, we can also adapt the feature extraction cost of the path through a CSTC tree, originally defined in (3.9), which becomes:

$$\sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{v^j \in \pi^l} \sum_{t=1}^{T} |F_{\alpha t}\beta_t^j| \right\|_0 \longrightarrow \sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t}\beta_t^j)^2}. \tag{3.22}$$

CART evaluation cost. The evaluation of a CART tree may be non-trivial or comparable to the cost of feature extraction and its cost must be accounted for. We define a constant $e_t \geq 0$, which captures the cost of the evaluation of the t^{th} CART tree. We can express this evaluation cost for a single classifier with weight vector $\boldsymbol{\beta}$ in terms of the l_0 norm and again apply the mixed norm relaxation (3.12). The exact (left term) and relaxed evaluation cost penalty (right term) can be stated as follows:

$$\sum_{t=1}^{T} e_t \|\beta_t\|_0 \longrightarrow \sum_{t=1}^{T} e_t |\beta_t|$$

The left term incurs a cost of e_t for each tree h_t if and only if it is assigned a non-zero weight by the classifier, *i.e.* $\beta_t \neq 0$. Similar to feature values, we assume that CART tree evaluations can be cached and only incur a cost once (the first time they are computed). With this assumption, we can express the exact and relaxed CART evaluation cost along a path π^l in a CSTC tree as

$$\sum_{t=1}^{T} e_t \left\| \sum_{v^j \in \pi^l} |\beta_t^j| \right\|_0 \longrightarrow \sum_{t=1}^{T} e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}. \tag{3.23}$$

It is worth pointing out, that (3.23) is analogous to the feature extraction cost with linear classifiers (3.9) and its relaxation, as stated in (3.13).

CSTC and CSCC with non-linear classifiers. We can integrate the two CART tree aware cost terms (3.22) and (3.23) into the optimization problem in (3.13). The final objective of the CSTC tree after the "boosting trick" becomes then

$$\sum_{v^k \in \mathcal{V}} \underbrace{\left(\frac{1}{n} \sum_{i=1}^{n} p_i^k \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \sum_{v^l \in \mathcal{L}} p^l \Bigg[\underbrace{\sum_t e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}}_{\text{CART evaluation cost}} + \underbrace{\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}}_{\text{feature cost}} \Bigg]. \tag{3.24}$$

The objective in (3.24) can be optimized with the same block coordinate descent algorithm, as described in Section 3.4.3. Similarly, the CSCC loss function with non-linear classifiers becomes

$$\underbrace{\sum_{v^k \in \mathcal{V}} \left(\frac{1}{n} \sum_{i=1}^{n} \mu_{y_i} p_i^{k+1} \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \sum_{v^l \in L} p^l \Bigg[\underbrace{\sum_t e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}}_{\text{CART evaluation cost}} + \underbrace{\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}}_{\text{feature cost}} \Bigg]. \tag{3.25}$$

3.7 Results

In this section, we evaluate CSTC on a synthetic cost-sensitive learning task and compare it with competing algorithms on a large-scale, real world benchmark problem. Additionally, we discuss the differences between our models for several learning settings. We provide further insight by analyzing the features extracted on this dataset and looking at how CSTC tree partitions the input space.

3.7.1 Synthetic data

We construct a synthetic regression data set consisting of points sampled from the four quadrants of the X, Z-plane, where $X = Z \in [-1, 1]$. The features belong to two categories: cheap features: $sign(x), sign(z)$ with cost $c = 1$, which can be used to identify the quadrant of an input; and expensive features: $z_{++}, z_{+-}, z_{-+}, z_{--}$ with cost $c = 10$, which equal the exact label of an input if it is from the corresponding quadrant (or a random number otherwise).

Since in this synthetic data set we do not transform the feature space, we have $\mathbf{h}(\mathbf{x}) = \mathbf{x}$, and $\mathbf{F}$ (the weak learner feature-usage variable) is the 6×6 identity matrix. By design, a perfect classifier can use the two cheap features to identify the sub-region of an instance and then extract the correct expensive feature to make a perfect prediction. The minimum feature cost of such a perfect classifier is exactly $c = 12$ per instance. We construct the data set to be a regression problem, with labels sampled from Gaussian distributions with quadrant-specific means $\mu_{++}, \mu_{-+}, \mu_{+-}, \mu_{--}$ and variance 1. The individual values for the label means are picked to satisfy the CSTC assumption, *i.e.* that the prediction of similar labels requires similar features. In particular, as can be seen in Figure 3.4 (bottom left), label means from quadrants with negative $z-$coordinates (μ_{+-}, μ_{--}) are higher than those with positive $z-$coordinates (μ_{++}, μ_{-+}).

Figure 3.4 shows the raw data (bottom left) and a CSTC tree trained on this data with its predictions of test inputs made by each node. In general, in every path along the tree, the first two classifiers split on the two cheap features and identify the correct sub-region of the input. The leaf classifiers extract a single expensive feature to predict the labels. As such, the mean squared error of the training and testing data both approach zero at optimal cost $c = 12$.

3.7.2 Yahoo! Learning to Rank

To evaluate the performance of CSTC on real-world tasks, we test it on the Yahoo! Learning to Rank Challenge (LTR) data set. The set contains 19,944 queries and 473,134 documents. Each query-document pair $\mathbf{x}_i$ consists of 519 features. An extraction cost, which takes on a value in the set $\{1, 5, 20, 50, 100, 150, 200\}$, is associated with each feature[5]. The unit of these values turns out to be approximately the number of weak learner evaluations $h_t(\cdot)$ that can be performed while the feature is being extracted. The label $y_i \in \{4, 3, 2, 1, 0\}$ denotes the relevance of a document to its corresponding query, with 4 indicating a perfect match. We measure the performance using normalized discounted cumulative gain at the 5^{th} position (NDCG@5) [58], a preferred ranking metric when multiple levels of relevance are available. To introduce non-linearity, we transform the input features into a non-linear feature space $\mathbf{x} \to \mathbf{h}(\mathbf{x})$ with the boosting trick (see Section 1.4.1 and Section 3.6) with $T = 3000$ iterations

[5]The extraction costs were provided by a Yahoo! employee.

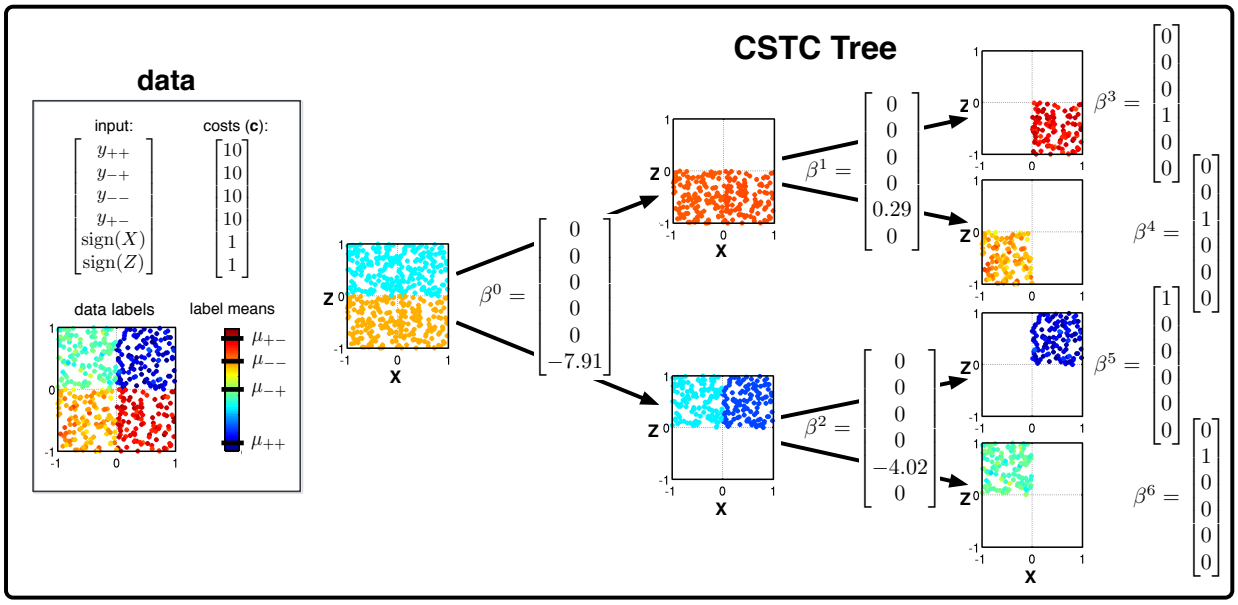

Figure 3.4: CSTC on synthetic data. The box at left describes the data set. The rest of the figure shows the trained CSTC tree. At each node we show a plot of the predictions made by that classifier and the feature weight vector. The tree obtains a perfect (0%) test-error at the optimal cost of 12 units.

of gradient boosting and CART trees of maximum depth 4. Unless otherwise stated, we determine the CSTC depth by validation performance (with a maximum depth of 10).

Figure 3.5 shows a comparison of CSTC with several recent algorithms for learning under test-time resource constraints. We show NDCG versus cost (in units of weak learner evaluations). We obtain the curves of CSTC by varying the accuracy/cost trade-off parameter λ (and perform early stopping based on the validation data, for fine-tuning). For CSTC we evaluate eight settings, $\lambda = \{\frac{1}{3}, \frac{1}{2}, 1, 2, 3, 4, 5, 6\}$. In the case of *stage-wise regression*, which is not cost-sensitive, the curve is simply a function of the number of boosting iterations. We include CSTC with and without fine-tuning. The comparison shows that there is a small but consistent benefit to fine-tuning the weights as described in Section 3.4 (*Fine-tuning*).

For competing algorithms, we include *Early exit* [14] which improves upon stage-wise regression by short-circuiting the evaluation of unpromising documents at test-time, reducing the overall test-time cost. The authors propose several criteria for rejecting inputs early and we use the best-performing method "early exits using proximity threshold", where at the

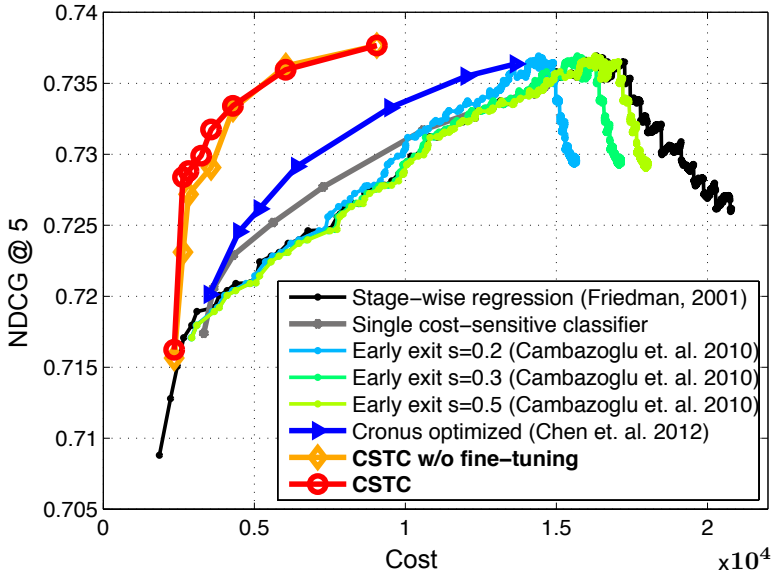

Figure 3.5: The test ranking accuracy (NDCG@5) and cost of various cost-sensitive classifiers. CSTC maintains its high retrieval accuracy significantly longer as the cost-budget is reduced.

i^{th} early-exit, we remove all test-inputs that have a score that is at least $\frac{300-i}{299}s$ lower than the fifth best input, and s determines the power of the early-exit. The *single cost-sensitive classifier* is a trivial CSTC tree consisting of only the root node *i.e.* a cost-sensitive classifier without the tree structure. We also include *Cronus* [24]. Setting the maximum number of Cronus nodes to 10 and setting all other parameters (*eg.* keep ratio, discount, early-stopping) based on a validation set. As shown in the graph, both Cronus and CSTC improve the cost/accuracy trade-off curve over all other algorithms. The power of Early exit is limited in this case as the test-time cost is dominated by feature extraction, rather than the evaluation cost. Compared with Cronus, CSTC has the ability to identify features that are most beneficial to different groups of inputs. It is this ability which allows CSTC to maintain the high NDCG significantly longer as the cost-budget is reduced. It is interesting to observe that the single cost-sensitive classifier outperforms stage-wise regression (due to the cost sensitive regularization) but obtains much worse cost/accuracy trade-offs than the full CSTC tree. This demonstrates that the tree structure is indeed an important part of the high cost effectiveness of CSTC.

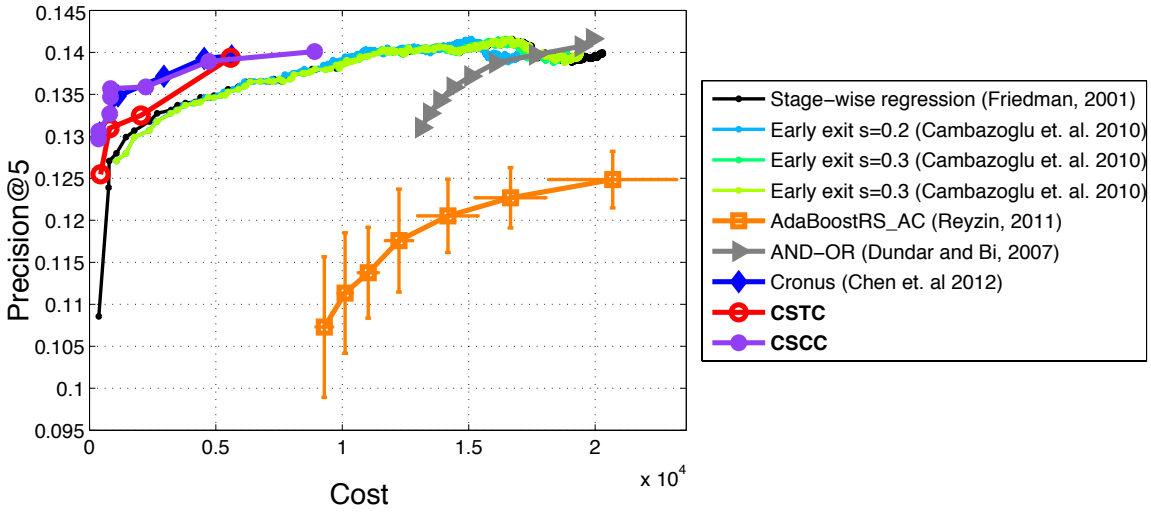

Figure 3.6: The test ranking accuracy (Precision@5) and cost of various cascade classifiers on the LTR-Skewed data set with high class imbalance. CSCC outperforms similar techniques, requiring less cost to achieve the same performance.

3.7.3 Yahoo! Learning to Rank: Skewed, Binary

To evaluate the performance of our cascade approach CSCC, we construct a highly class-skewed binary data set using the Yahoo! LTR data set. We binarize the data by defining inputs having labels $y_i \geq 3$ as "relevant" and label the rest as "irrelevant". We also replicate each negative, irrelevant example 10 times to simulate the scenario where only a few documents are highly relevant, out of many candidate documents. After these modifications, the inputs have one of two labels $\{-1, 1\}$, and the ratio of $+1$ to -1 is $1/100$. We call this data set LTR-Skewed. This simulates an important setting, as in many time-sensitive real life applications the class distributions are often very skewed.

For the binary case, we use the ranking metric Precision@5 (the fraction of top 5 documents retrieved that are relevant to a query). It best reflects the capability of a classifier to precisely retrieve a small number of relevant instances within a large set of irrelevant documents. Figure 3.6 compares CSCC and CSTC with several recent algorithms for learning under test-time resource constraints, and specifically for binary classification. We show Precision@5 versus cost (in units of weak learner evaluations). Similar to CSTC, we obtain the curves

of CSCC by varying the accuracy/cost trade-off parameter λ. For CSCC we evaluate eight settings, $\lambda = \{\frac{1}{3}, \frac{1}{2}, 1, 2, 3, 4, 5, 6\}$.

For competing algorithms, in addition to *Early exit* [14] and *Cronus* [24] described above, we also include *AND-OR* proposed by Dundar and Bi [37], which is designed specifically for binary classification. They formulate a global optimization of a cascade of classifiers and employ an *AND-OR* scheme with the loss function that treats negative inputs and positive inputs separately. This setup is based on the insight that positive inputs are carried all the way through the cascade (*i.e.* each classifier must classify them as positive), whereas negative inputs can be rejected at any time (*i.e.* it is sufficient if a single classifier classifies them as negative). The loss for positive inputs is the maximum loss across all stages, which corresponds to the AND operation, and encourages all classifiers to make correct predictions. For negative inputs the loss is the minimum loss of all classifiers, which corresponds to the OR operation, and which enforces that at least one classifier makes a correct prediction. Different from our approach, their algorithm requires pre-assigning features to each node. We therefore use five nodes in total, assigning features of cost $\leq 5, \leq 20, \leq 50, \leq 150, \leq 200$. The curve is generated by varying a loss/cost trade-off parameter (similar to λ). Finally, we also compare with the cost sensitive version of *AdaboostRS* [100]. This algorithm resamples decision trees, learned with AdaBoost [42], inversely proportional to a tree's feature cost. As this algorithm involves random sampling, we averaged over 10 runs and show the standard deviations in both precision and cost.

As shown in the graph, AdaBoostRS obtains lower precision than other algorithms. This may be due to the known sensitivity of AdaBoost towards noisy data [82]. AND-OR also underperforms. It requires pre-assigning features prior to training, which makes it impossible to obtain high precision at a low cost. On the other hand, Cronus, CSCC, and CSTC have the ability to cherry pick good but expensive features at an early node, which in turn can reduce the overall cost while improving performance over other algorithms. We take a closer look at this effect in the following section. Cronus and CSCC in general outperform CSTC because they can exit a large portion of the dataset early on. As mentioned before, CSCC slightly outperforms Cronus, which we attribute to the more principled optimization.

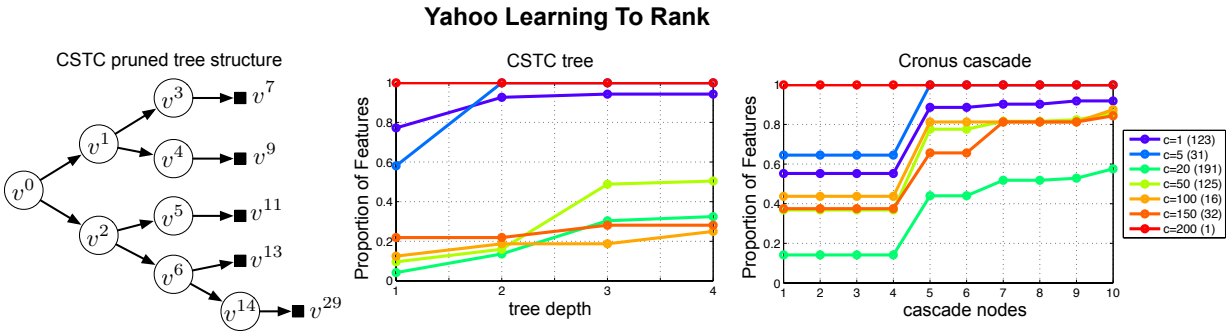

Figure 3.7: *Left:* The pruned CSTC tree, trained on the Yahoo! LTR data set. The ratio of features, grouped by cost, are shown for CSTC (*center*) and Cronus (*right*). The number of features in each cost group is indicated in parentheses in the legend. More expensive features ($c \geq 20$) are gradually extracted deeper in the structure of each algorithm.

3.7.4 Feature extraction

Based on the LTR and LTR-Skewed data sets, we investigate the features extracted by various algorithms in each scenario. We first show the features retrieved in the regular balanced class data set (LTR). Figure 3.7 (*left*) shows the pruned CSTC tree learned on the LTR data set. The plot in the center demonstrates the fraction of features, with a particular cost, extracted at different depths of the CSTC tree. The rightmost plot shows the features extracted at different nodes of Cronus. We observe a general trend that for both CSTC and Cronus, as depth increases, more features are being used. However, cheap features ($c \leq 5$) are all extracted early-on, whereas expensive features ($c \geq 20$) are extracted by classifiers sitting deeper in the tree. Here, individual classifiers only cope with a small subset of inputs and the expensive features are used to classify these subsets more precisely. The only feature that has cost 200 is extracted at all depths—which seems essential to obtain high NDCG [24]. Although Cronus has larger depth than CSTC (10 vs 4), most nodes in Cronus are basically dummy nodes (as can be seen by the flat parts of the feature usage curve). For these nodes all weights are zeros, and the threshold is a very small negative number, allowing all inputs to pass through.

In the second scenario, where the class-labels are binarized and are highly skewed (LTR-Skewed), we compare the features extracted by CSCC, Cronus and AND-OR. For a fair comparison, we set the trade-off parameter λ for each algorithm to achieve similar precision

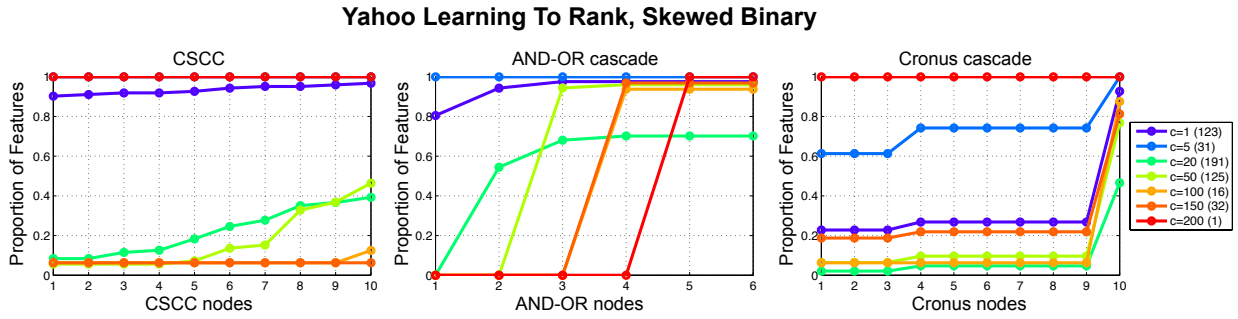

Figure 3.8: The ratio of features, grouped by cost, that are extracted at different depths of CSCC (*left*), AND-OR (*center*) and Cronus (*right*). The number of features in each cost group is indicated in parentheses in the legend.

0.135 ± 0.001. We also set the maximum number of nodes of CSCC and Cronus to 10. Figure 3.8 (*left*) shows the fraction of features, with a particular cost, extracted at different nodes of the CSCC. The center plot illustrates the features used by AND-OR, and the right plot shows the features extracted at different nodes of Cronus. Note that while the features are pre-assigned in the AND-OR algorithm, it still has the ability to only use some of the assigned features at each node. In general, all algorithms use more features as the depth increases. However, compared to AND-OR, both Cronus and CSCC can cherry pick some good but expensive features early-on to achieve high accuracy at a low cost. Some of the expensive features (e.g., $c = 100, 150$) are extracted from the very first node in CSCC and Cronus, whereas in AND-OR they are only available at the fourth node. This ability is one of the reasons that CSCC and Cronus achieve better performance over existing cascade algorithms.

3.7.5 Input space partition

CSTC has the ability to split the input space and learn more specialized classifiers sitting deeper in the tree. Figure 3.9 (*left*) shows a pruned CSTC tree ($\lambda = 4$) for the LTR data set. The number above each node indicates the average label of the testing inputs passing through that node. We can observe that different branches aim at different parts of the input domain. In general, the upper branches focus on correctly classifying higher-ranked documents, while the lower branches target low-ranked documents. Figure 3.9 (*right*) shows

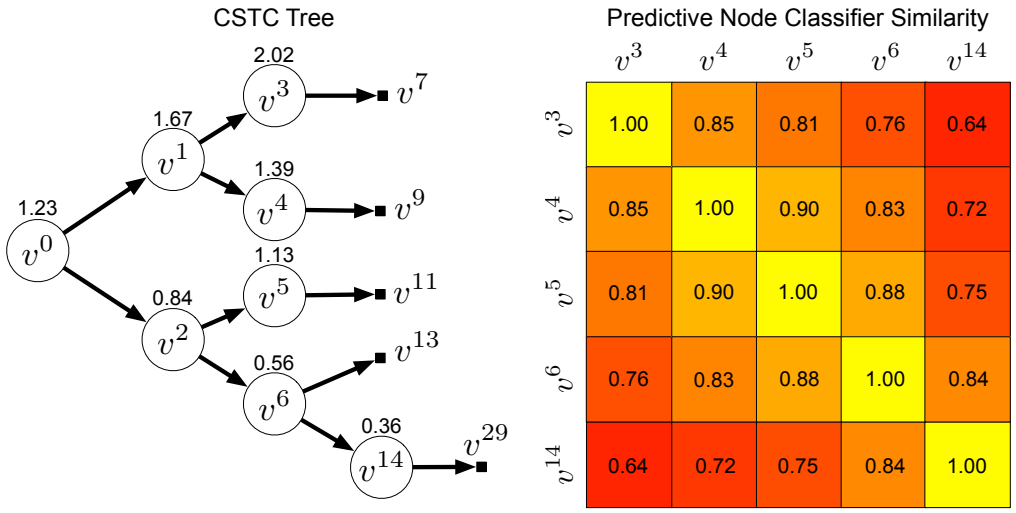

Figure 3.9: (*Left*) The pruned CSTC-tree generated from the Yahoo! Learning to Rank data set. (*Right*) Jaccard similarity coefficient between classifiers within the learned CSTC tree.

the Jaccard matrix of the leaf classifiers $(v^3, v^4, v^5, v^6, v^{14})$ from this CSTC tree. The number in field i, j indicates the fraction of shared features between v^i and v^j. The matrix shows a clear trend that the Jaccard coefficients decrease monotonically away from the diagonal. This indicates that classifiers share fewer features in common if their average labels are further apart—the most different classifiers v^3 and v^{14} have only 64% of their features in common—and validates that classifiers in the CSTC tree extract different features in different regions of the tree.

3.8 Conclusion

In this chapter, we propose a novel algorithm that aims to reduce amortized test-time cost. We formulate the test-time cost and accuracy trade-off systematically into a tree of classifiers and relax it into a well-behaved optimization problem. The resulting algorithm, Cost-Sensitive Tree of Classifiers (CSTC), partitions the input space into sub-regions and identifies the most cost-effective features for each one of these regions—allowing it to match the high

accuracy of the state-of-the-art at a small fraction of the cost. We further extend the algorithm into a Cost-sensitive Cascade of Classifiers (CSCC) to deal with binary imbalanced data sets and achieve the state-of-the-art test-time cost/accuracy performance.

Chapter 4

Model Compression

In this chapter, we switch our attention from feature extraction cost to classifier evaluation cost. In previous chapters, we assume that classifier evaluation cost is very low compared to feature extraction cost. However, if the classifier is nonparametric, its model (*i.e.* support vectors) could be very large when data size is large, and as a result, its evaluation cost could be substantial. To reduce classifier evaluation cost of nonparametric classifiers, we introduce a new strategy: model compression. Based on this strategy, we propose a novel algorithm, *Compressed Vector Machine (CVM)*. CVM focuses specifically on kernel support vector machines (SVM) and compresses the learned kernel SVM model by re-solving the exact SVM optimization problem to cherry-pick a small subset of support vectors and optimize them. By *moving* these selected support vectors to approximate the decision boundary learned from the full model, CVM achieves a relatively high classification accuracy with much fewer support vectors, and greatly reduces the classifier evaluation cost.

4.1 Introduction

Support Vector Machines (SVM) are arguably one of the great success stories of machine learning and have been used in many real world applications, including email spam classification [35], face recognition [54] and gene selection [52]. SVM can classify data sets along highly non-linear decision boundaries because of the kernel-trick [110]. However, different from the classifiers we discussed in previous chapters, the expressiveness of SVMs comes at a price: kernel SVM is a nonparametric model, and during test-time, the SVM needs to compute the kernel function of a test sample and its model parameters (*i.e.* support vectors).

This computation is very expensive: First, it is linear in the number of support vectors, and second, it often requires expensive exponentiation (*e.g.* for the radial basis or χ^2 kernels). This expensive evaluation cost is particularly prominent in settings with strong resource constraints (*e.g.* embedded devices, cell phones or tablets) or frequently repeated tasks (*e.g.* webmail spam classification, web-search ranking, face detection in uploaded images), which can be performed billions of times per day.

To budget the test-time evaluation cost, we describe an approach that does not *select* support vectors from the training set, but instead *learns* them to match a pre-defined SVM decision boundary. Given an existing SVM model with r support vectors, it learns $m \ll r$ "artificial support vectors", which are not originally part of the training set. The resulting model is a standard SVM classifier (thus can be saved, for example, in a LibSVM [16] compatible file). Relative to the original model, it has comparable accuracy, but it is up to several orders of magnitudes smaller and faster to evaluate. We refer to our algorithm as *Compressed Vector Machine (CVM)* and demonstrate on six real-world data sets of various size and complexity that it achieves unmatched accuracy vs. test-time cost trade offs.

4.2 Related Work

Reducing test-time cost has recently attracted much attention. A lot of work [13, 24, 50, 74, 96, 105, 126, 134] focus on scenarios where features are extracted on-demand and the extraction cost dominates the overall test-time cost. Their objective is to minimize the feature extraction cost.

Model compression was pioneered by Bucilu et al. [10]. Our work was inspired by their vision, however it differs substantially, as we do not focus on ensembles of classifiers and instead learn a model compressor explicitly for SVMs. More recently, Xu et al. [135] introduces an algorithm to reduce the test-time cost specifically for the SVM classifier. However, they focus on learning a new representations consisting of cheap non-linear features for linear SVMs.

Dekel et al. [32] propose an algorithm to limit the memory usage for kernel based online classification. Different from our approach, their algorithm is not a post-process procedure, and instead they modify the kernel function directly to limit the amount of memory the

algorithm uses. Similar to [32], Wang et al. [127] also focuses on online kernel SVM, and attacks primarily the training time complexity. Post-process algorithms [26, 25] are widely used to approximate solutions in large scale applications. However, these algorithms do not solve the exact same original optimization problem and focus exclusively on manifold learning.

Keerthi et al. [64] propose a similar approach to our work. They specifically reduce the SVM evaluation cost by reducing the number of support vectors. Heuristics are used to select a small subset of support vectors, up to a given budget, during training time, thus solving an approximate SVM optimization. However, different from their approach, our method is a post-processing compression to the regular SVM. We begin from an exact SVM solution and compresses the set of support vectors by choosing and optimizing over a small subset of support vectors to approximate the optimal decision boundary. This post-processing optimization framework renders unmatched accuracy and cost performance.

Perhaps the most relevant work is proposed by Burges [11]. They also learn a reduced set of artificial support vectors to approximate the decision boundary of the full SVM model. However, different from ours, they initialize these artificial support vectors randomly, whereas we carefully select a compressed set of support vectors by re-solving the exactly original SVM optimization problem with stage-wise regression.

4.3 Background

In this section, we briefly re-cap kernel support vector machines and introduce an useful forward selection method used in our algorithm.

Kernel support vector machines. Given instances $\mathbf{x}$ in a low dimensional space, the *kernel-trick* [110] maps the original feature space $\mathbf{x}$ into a higher (possibly infinite) dimensional space $\phi(\mathbf{x})$. SVMs learn a hyperplane in this higher dimensional space by maximizing the margin $\frac{1}{\|\mathbf{w}\|}$ and penalizing training instances on the wrong side of the hyperplane,

$$\min_{\mathbf{w},b} \|\mathbf{w}\| + C \sum_{i}^{n} \Big(\max \big(1 - y_i \mathbf{w}^\top \phi(\mathbf{x}_i) + b, 0\big) \Big)^2, \tag{4.1}$$

where b is the bias, and C trades-off regularization/margin and training accuracy. Note that we use the quadratic hinge loss penalty and thus (4.1) is differentiable. The power of the kernel trick is that the higher dimensional space $\phi(\mathbf{x})$ never needs to be expressed explicitly, because (4.1) can be formulated in terms of inner products between input vectors in the higher dimensional space. Let a matrix $\mathbf{K}$ denote these inner products, where $\mathbf{K}_{ij} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$, and $\mathbf{K}$ is the training *kernel matrix*. The optimization in (4.1) can be then expressed in terms of the kernel matrix $\mathbf{K}$ and coefficients $\boldsymbol{\alpha}$ using the *representer theorem* [65, 17]:

$$\min_{\boldsymbol{\alpha},b} \Big(\max(\mathbf{1} - \hat{\mathbf{K}}\boldsymbol{\alpha} - \mathbf{y}b, 0) \Big)^2 + \boldsymbol{\alpha}^\top \mathbf{K} \boldsymbol{\alpha}. \tag{4.2}$$

The classification rule $J(\cdot)$ for a test input $\mathbf{x}_t$ can also be expressed by testing kernel $\tilde{\mathbf{K}}$ that consists of inner products between a test input $\mathbf{x}_t$ and support vectors $\mathcal{S} = \{\mathbf{x}_i | \alpha_i \neq 0\}$, $\tilde{\mathbf{K}}_{it} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_t)$, where

$$J(\phi(\mathbf{x}_t)) = \sum_{i=1}^{n} \alpha_i y_i \tilde{\mathbf{K}}_{it} + b. \tag{4.3}$$

Note that once the testing kernel $\tilde{\mathbf{K}}$ is computed, generating the final prediction is merely a linear combination, and thus the dominating cost is computing the testing kernel itself.

Least angle regression (LARS). LARS [38] is a widely used forward selection algorithm because of its simplicity and efficiency. Given input vectors $\mathbf{x}$, target labels $\mathbf{y}$, and a quadratic loss $\ell(\boldsymbol{\beta}) = (\mathbf{x}^\top \boldsymbol{\beta} - \mathbf{y})^2$, LARS learns to approximate labels by building up the coefficient vector $\boldsymbol{\beta}$ in successive steps, starting from an all-zero vector. To minimize the loss function ℓ, LARS initially descends on a coordinate direction that has the largest gradient,

$$\beta_t^* = \operatorname*{argmax}_{\beta_t} \frac{\partial \ell}{\partial \beta_t}. \tag{4.4}$$

The algorithm then incorporates this coordinate into its active set. After identifying the gradient direction, LARS selects the step size very carefully. Instead of too greedy or too tiny, LARS computes a step size such that after taking this step, a new direction *outside* of the active set has the same maximum gradient as directions *in* the active set. LARS then include this new direction into the active set.

In the following iterations, LARS gradient descends on a direction that maintains the same gradient for all directions in the active set. In other words, LARS descends following an *equiangular* direction of all directions in the active set. The algorithm then repeats computing step-size, including new directions into the active set, and descending on an equiangular directions. This process makes LARS very efficient, as after T iterations, LARS solution has exactly T directions in the active set, or equivalently, only T non-zero coefficients in $\boldsymbol{\beta}$.

4.4 Method

In this section, we detail our approach to reduce the test-time SVM evaluation cost. We regard our approach as a post-processing compression to the original SVM solution. After solving an SVM, we obtain a set of support vectors $\mathcal{S} = \{\mathbf{x}_i | \, \alpha_i \neq 0\}$, and the corresponding coefficients $\boldsymbol{\alpha}$. Given the original SVM solution, we can model the test-time evaluation cost explicitly.

Kernel SVM evaluation cost. Based on the prediction function (4.3) we can formulate the exact SVM classifier evaluation cost. Let e denote the cost of computing a test kernel entry $\tilde{\mathbf{K}}_{it}$ (*i.e.* kernel function of a test input $\mathbf{x}_t$ and a support vector $\mathbf{x}_i$). We assume that the computation cost is identical across all test inputs and all support vectors. As shown in (4.3), generating a prediction for a test input requires computing the kernel entries between the test input and all support vectors. The total evaluation cost is therefore a function of the number of support vectors n_{sv}. After obtaining all kernel entries for a test point $\mathbf{x}_t$, the final prediction is simply linear combination of the kernel row $\tilde{\mathbf{K}}_t$ weighted by $\boldsymbol{\alpha}$. The cost of computing this linear combination is very low compared to the kernel computation, and therefore the total evaluation cost $c_e \approx n_{\text{sv}} e$. We aim to reduce the size of the support vector set n_{sv} without greatly affecting prediction accuracy.

Removing non-support vectors. Since the test-time evaluation cost is a function of the number of support vectors, the goal is to cherry-pick and optimize a subset of the optimal support vectors bounded in size by a user-specified compression ratio. We first note that all non-support vectors can be removed during this process without affecting the full SVM solution. If we define a design matrix $\hat{\mathbf{K}} \in \mathcal{R}^{n \times n}$, where $\hat{\mathbf{K}}_{ij} = y_i \mathbf{K}_{ij}$, and note that the weight vector in (4.1) can be expressed as $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \phi(\mathbf{x}_i)$. The squared penalty SVM

objective function in (4.1) can be expressed with coefficients $\boldsymbol{\alpha}$ and the kernel matrix $\mathbf{K}$ using the *representer theorem* [65, 17]:

$$\min_{\boldsymbol{\alpha},b} \Big(\max(\mathbf{1} - \hat{\mathbf{K}}\boldsymbol{\alpha} - \mathbf{y}b, 0)\Big)^2 + \boldsymbol{\alpha}^\top \mathbf{K}\boldsymbol{\alpha}. \tag{4.5}$$

Since (4.5) is a *strongly* convex function, and all non-support vectors have the corresponding coefficient $\alpha_i = 0$, we can remove all non-support vectors from the optimization problem and the full SVM optimal solution stays the same.

To find an optimal subset of support vectors given the compression ratio, we re-train the SVM with only support vectors and a constraint on the number of support vectors. Note that $\boldsymbol{\alpha}$ are effectively the coefficients of support vectors, and we can efficiently control the number of support vectors by adding an l_0-norm on $\boldsymbol{\alpha}$, where the l_0-norm effectively counts the number of non-zeros in $\boldsymbol{\alpha}$. The optimization problem becomes

$$\begin{aligned} &\min_{\boldsymbol{\alpha},b} \Big(\mathbf{1} - \hat{\mathbf{K}}\boldsymbol{\alpha} - \mathbf{y}b\Big)^2 + \boldsymbol{\alpha}^\top \mathbf{K}\boldsymbol{\alpha} \\ &\text{s.t.} \|\boldsymbol{\alpha}\|_0 \leq \frac{1}{e} B_e, \end{aligned} \tag{4.6}$$

where B_e is evaluation cost budget, and consequently, $\frac{1}{e}B_e$ is the desired number of support vectors based on the budget. Note that after removing non-support vectors, we obtain a condensed matrix $\mathbf{K}$ and $\hat{\mathbf{K}} \in \mathcal{R}^{n_{sv} \times n_{sv}}$, and $\mathbf{y} \in \{-1, +1\}^{n_{sv}}$.

Forming ordinary least squares problem. The current form of equation (4.6) can be made more amenable to optimization by rewriting the objective function as an ordinary least square problem. Expanding the squared term, simplifying, and fixing the bias term b (as it does not affect the solution dramatically), we re-format the objective function (4.6) into

$$\min_{\boldsymbol{\alpha}} (\mathbf{1} - \mathbf{y}b)^\top (\mathbf{1} - \mathbf{y}b) - 2\boldsymbol{\alpha}^\top \hat{\mathbf{K}}^\top (\mathbf{1} - \mathbf{y}b) + \boldsymbol{\alpha}^\top (\hat{\mathbf{K}}^\top \hat{\mathbf{K}} + \mathbf{K})\boldsymbol{\alpha}. \tag{4.7}$$

We introduce two auxiliary variables Ω and $\boldsymbol{\beta}$, where $\Omega^\top \Omega = \hat{\mathbf{K}}^\top \hat{\mathbf{K}} + \mathbf{K}$ and $\Omega^\top \boldsymbol{\beta} = -\hat{\mathbf{K}}^\top (\mathbf{1} - \mathbf{y}b)$. Because $\hat{\mathbf{K}}^\top \hat{\mathbf{K}} + \mathbf{K}$ is a symmetric matrix, we can compute its eigen-decomposition

$$\hat{\mathbf{K}}^\top \hat{\mathbf{K}} + \mathbf{K} = \mathbf{S}\mathbf{D}\mathbf{S}^\top, \tag{4.8}$$

where $\mathbf{D}$ is the diagonal matrix of eigenvalues and $\mathbf{S}$ is the orthonormal matrix of eigenvectors. Moreover, because the matrix $\hat{\mathbf{K}}^\top\hat{\mathbf{K}} + \mathbf{K}$ is positive semi-definite, we can further decompose $\mathbf{SDS}^\top$ into an inner product of two real matrices by taking the square root of $\mathbf{D}$. Let $\Omega = \sqrt{\mathbf{D}}\mathbf{S}^\top$, and we obtain a matrix Ω that satisfies $\Omega^\top\Omega = \hat{\mathbf{K}}^\top\hat{\mathbf{K}} + \mathbf{K}$. After computing Ω, we can readily compute $\boldsymbol{\beta} = -(\Omega^\top)^{-1}\hat{\mathbf{K}}^\top(\mathbf{1} - \mathbf{y}b)$, where $(\Omega^\top)^{-1} = \frac{1}{\sqrt{\mathbf{D}}}\mathbf{S}^\top$.

With the help of the two auxiliary variables, we convert (4.7), plus a constant term[6], into least squares format. Together with relaxation of the non-continuous l_0-norm constraint to an l_1-norm constraint, we obtain

$$\min_{\boldsymbol{\alpha}} (\Omega\boldsymbol{\alpha} + \boldsymbol{\beta})^2, \quad \text{s.t. } \|\boldsymbol{\alpha}\|_1 \leq \frac{1}{e}B'_e, \tag{4.9}$$

where B'_e is used to relax the constraint B_e resulting from using l_1-norm to approximate l_0-norm in (4.6).

Compressing the support vector set. The squared loss and l_1 constraint in (4.9) lead naturally to the LARS algorithm. Given a budget B_e, we can determine the maximum size m of the compressed support vector set ($m = \frac{B_e}{e}$). Using LARS, we start from an empty support vector set and add m support vectors incrementally. Since adding a support vector is equivalent to activating a coefficient in $\boldsymbol{\alpha}$ to a non-zero value, we can obtain m optimal support vectors by running LARS optimization in (4.9) exactly m steps, where each step activates one coefficient. The resulting solution gives the optimal set of m support vectors. We refer this intermediate step as *LARS-SVM*. Note that this step is crucial for the problem, as this LARS-SVM solution is obtained by solving the exact SVM optimization problem, and it serves as a very good initialization for the next step, which is a non-convex optimization problem.

Gradient support vectors. If we interpret $\boldsymbol{\alpha}$ as coordinates and the corresponding columns in the kernel matrix $\mathbf{K}$ as basis vectors, then these basis vectors span an $\mathcal{R}^{n_{sv}}$ space in which lie predictions of the original SVM model. In this compression algorithm, our goal is to find a lower dimensional space that supports good approximations of the original predictions. After running LARS for m iterations, we obtain m support vectors and their coefficients $\boldsymbol{\alpha}$, forming an $\mathcal{R}^m$ affine space of the space spanned by the full kernel matrix.

[6] $(\mathbf{1} - \mathbf{y}b)^\top \left(\hat{\mathbf{K}}(\hat{\mathbf{K}}^\top\hat{\mathbf{K}} + \mathbf{K})^{-1}\hat{\mathbf{K}}^\top - \mathbf{I}\right)(\mathbf{1} - \mathbf{y}b)$

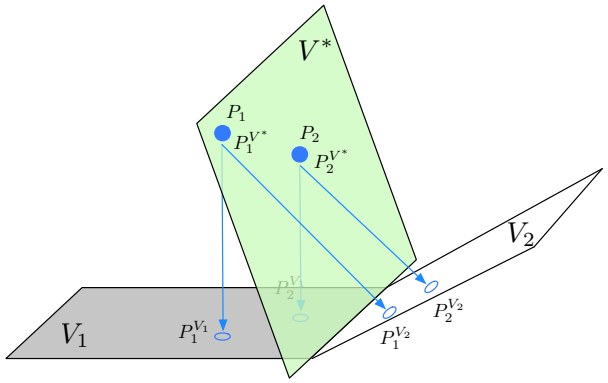

Figure 4.1: Illustration of searching for a space $V \in \mathcal{R}^2$ that best approximates predictions P_1 and P_2 of training instances in $\mathcal{R}^3$ space. Neither V_1 or V_2, spanned by existing columns in the kernel matrix, is a good approximation. V^* spanned by kernel columns computed from two *artificial* support vectors is the optimal solution.

We illustrate this lower dimensional approximation in Figure 4.1. Vectors P_1 and P_2 are predictions of two training points made in the full SVM solution space ($\mathcal{R}^3$ and spanned by three support vectors). We want to compress the model to two support vectors by looking for a space $V \in \mathcal{R}^2$ that supports the best approximations of these two predictions. Using existing support vectors as a basis, we can find spaces V_1 and V_2, each spanned by a pair of support vectors. The projections of P_1 and P_2 on plane V_1 ($P_1^{V_1}$ and $P_2^{V_1}$) are closest to the original predictions P_1 and P_2, and thus V_1 is a better approximation, compared to V_2. However, in this case, neither V_1 nor V_2 is a particularly good approximation. Suppose we remove the restriction of selecting a space spanned by *existing* basis vectors in the kernel matrix, instead optimizing the basis vectors to yield a more suitable space. In Figure 4.1, this is illustrated by the optimal space V^* which produces a better approximation to the target predictions.

Note that the basis vectors (columns of the kernel matrix) are parameterized by support vectors. By optimizing these underlying support vectors, we can search for a better low-dimensional space. We follow the approach from [11]. We first denote $\mathbf{K}_m$ as the training kernel matrix with only m columns corresponding to the support vectors chosen by LARS, and $\boldsymbol{\alpha}_m$ as the coefficients of these support vectors. We can then formulate the search for *artificial* support vector as an optimization problem. Specifically, we minimize a squared loss between approximate and full SVM predictions over all support vectors, and the parameters

are the subset of support vectors chosen by LARS and their corresponding coefficient $\boldsymbol{\alpha}$.

$$\min_{(\mathbf{x}_1,\ldots,\mathbf{x}_m,\boldsymbol{\alpha})} \mathcal{L} = \left(\mathbf{K}_m \alpha_m - \mathbf{K}\boldsymbol{\alpha}\right)^2. \tag{4.10}$$

$\mathbf{K}_{ij}$ is one entry of the kernel matrix $\mathbf{K}$, and for simplicity, we use radial basis function (RBF) kernel function ($\mathbf{K}_{ij} = e^{\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$). However, other kernel functions are equally suitable. The unconstrained optimization problem (4.10) can be solved by conjugate gradient descent with respect to the chosen m support vectors. Since $\boldsymbol{\alpha}$'s are the coordinates with respect to the basis, we optimize $\boldsymbol{\alpha}$ jointly with support vectors, which is faster than optimizing basis and solving coordinates alternatively. The gradients can be computed very efficiently using matrix operations. Since gradient descent on support vectors is equivalent to moving these support vectors in a continuous space, thereby generating m *new* support vectors, we refer to these newly generated support vectors as *gradient support vectors*. Because the optimization problem in (4.10) is non-convex with respect to $\mathbf{x}_i$, we initialize our algorithm with the basis $\mathbf{K}_m$ and coordinates α_m returned in the LARS-SVM solution. We denote this combined method of LARS-SVM and gradient support vectors as Compressed Vector Machine (CVM).

4.5 Results

In this section, we first demonstrate Compressed Vector Machine (CVM) on a synthetic data set to graphically illustrate each step in the algorithm. We then evaluate CVM on several real-world data sets.

Synthetic data set. The data set contains 600 sample inputs from two classes (red and blue), where each input contains two features. The blue inputs are sampled from a Gaussian distribution with mean at the origin and variance 1, and red inputs are sampled from a noisy circle surrounding the blue inputs. As shown in Figure 4.2(a), by design the data set is not linearly separable. For simplicity, we treat all inputs as training inputs. To evaluate CVM, we first learn an SVM with RBF kernel from the full training set. We plot the resulting optimal decision boundary in Figure 4.2(b) with a black curve. In total, the full model has 248 support vectors, and they are enlarged points in Figure 4.2(b).

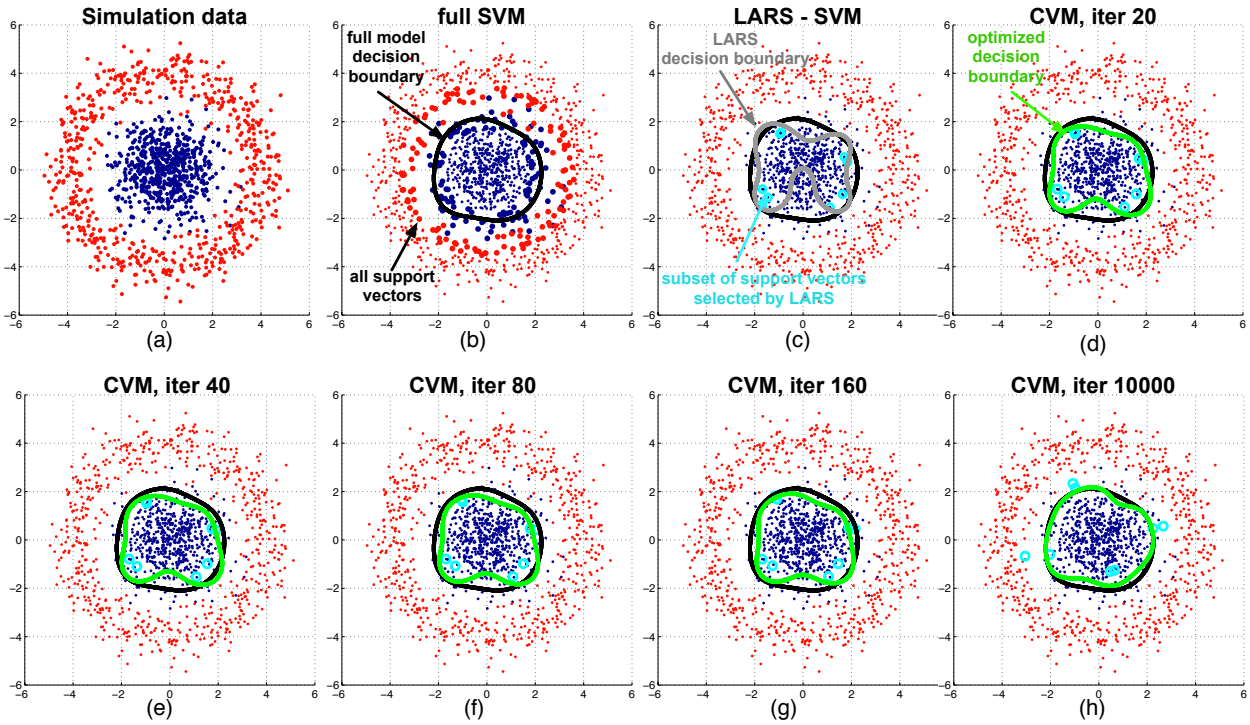

Figure 4.2: Illustration of each step of CVM on a synthetic data set. (a) Simulation inputs from two classes (red and blue). By design, the two classes are not linear separable. (b) Decision boundary formed by a full SVM solution (black curve), and all support vectors (enlarged points). (c) A small subset of support vectors picked by LARS (cyan circles) and the compressed decision boundary formed by this subset of support vectors (gray curve). (d-h) Optimization iterations. The gradient support vectors are moved by the iterative optimization. The optimized decision boundary formed by gradient support vectors (green curve) gradually approaches the one formed by the full SVM solution.

To compress the model, we first select a subset of support vectors by solving LARS-SVM optimization (4.9). Specifically, we compress the model to 3% of its original size, 8 support vectors, by running LARS for 8 iterations. The 8 LARS-SVM support vectors are shown in Figure 4.2(c) as circles in cyan color, and the approximate LARS-SVM decision boundary is shown by the gray curve.

Since the space formed by 8 support vectors is heavily restricted by the discrete training input space, the approximation is poor. To overcome this problem, we search for a better space or basis in a continuous space, and perform gradient descent on support vectors by optimizing (4.10). In Figure 4.2(d-h), we illustrate the optimization with updated support

Statistics	Pageblocks	Magic	Letters	20news	MNIST	DMOZ
#training exam.	4379	15216	16000	11269	60000	7184
#testing exam.	1094	3804	4000	7505	10000	1796
#features	10	10	16	200	784	16498
#classes	2	2	26	20	10	16

Table 4.1: Statistics of all six data sets.

vector locations and optimized decision boundaries as we gradually increase the number of iterations. The resulting *gradient support vectors* are shown as cyan circles and the new optimized decision boundaries formed from these new gradient support vectors are shown by green curves. After 10000 iterations, as shown in Figure 4.2(h), we can observe that the optimized decision boundary (green) is very close to the boundary captured in the full model (black). These optimized decision boundaries demonstrate that moving a small subset of support vectors in a continuous space can efficiently approximate the optimal decision boundary formed by the full SVM solution, supporting effective SVM model compression. Note that we use the simple gradient descent method for illustration, and in practice, conjugate gradient descent method can be used to decrease the number of iterations to converge.

Real-world data sets. To evaluate the performance of CVM on real-world applications, we evaluate our algorithm on six data sets of varying size, dimensionality and complexity. Table 4.1 details the statistics of all six data sets. We use LibSVM [16] to train a regular RBF kernel SVM using regularization parameter C and RBF kernel width σ selected on a 20% validation split. For multi-class data sets, we use the one-vs-one multi-class scheme. The classification accuracy of test predictions from this SVM model serves as a baseline in Figure 4.3 (full SVM).

Given the full SVM solution, we run CVM in two steps. First, we use LARS to solve the optimization problem in (4.9) using all support vectors from the original SVM model. An initial compressed support vector set is selected with a target compressed size (*e.g.* 10 out of 500 support vectors). The selected support vectors serve as the second baseline in Figure 4.3 (LARS-SVM). Second, we shift these support vectors in a continuous space by optimizing (4.10) w.r.t. the input support vectors and the corresponding coefficient $\boldsymbol{\alpha}$, generating gradient support vectors. This final set of gradient support vectors constitutes the CVM model. To show the trend of accuracy/cost performance, we plot the classification

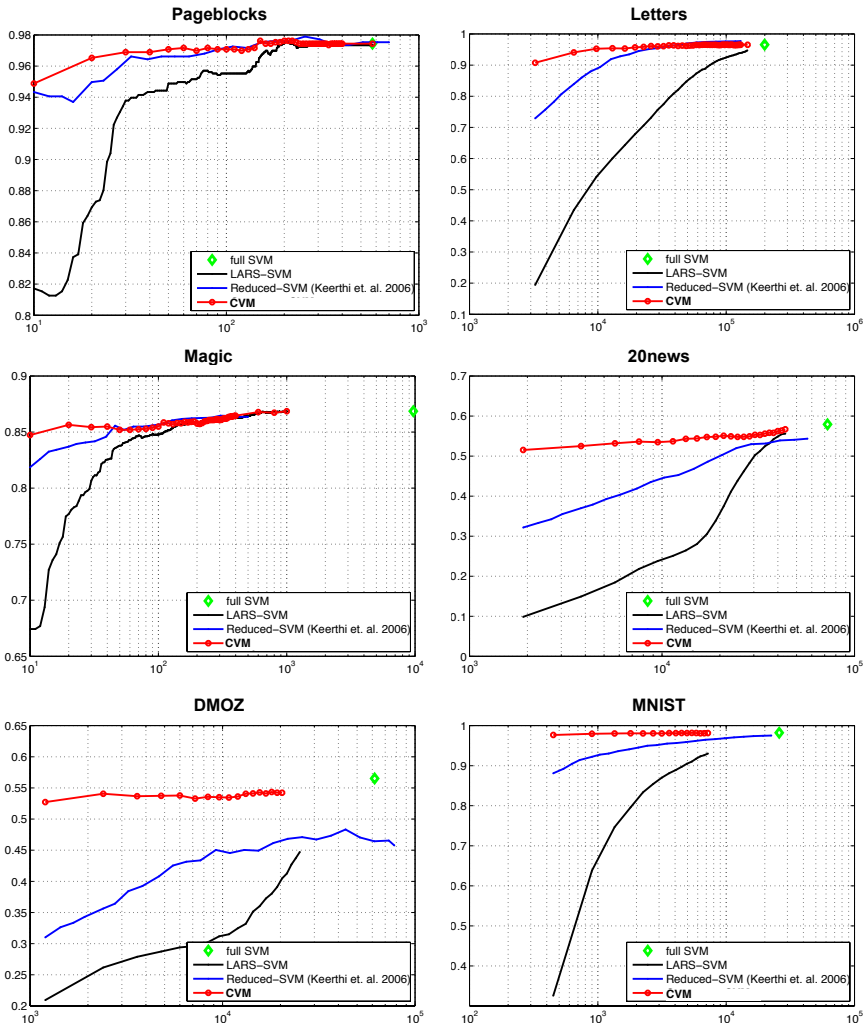

Figure 4.3: Accuracy versus number of support vectors (in log scale).

accuracy for CVM after adding every 10 support vectors. Figure 4.3 shows the performance of CVM and the baselines on all six data sets.

Comparison with prior work. Figure 4.3 also shows a comparison of CVM with *Reduced-SVM* [64]. This algorithm takes an iterative two phase approach. First a set of support vectors is heuristically selected from random samples of the training set and added to the existing set of support vectors (initially empty). Then, the model weights are optimized by an SVM with the quadratic hinge loss. The algorithm alternates these two steps until the target number of support vectors is reached.

As shown in Figure 4.3, CVM significantly improves over all baselines. Compared to the current state-of-the-art, Reduced-SVM, CVM has the capability of moving support vectors, generating a new basis, and learning a highly approximated basis to match the decision boundaries formed by the full SVM solution. It is this ability that distinguishes CVM from other algorithms when the evaluation budget is low. On all six data sets, CVM maintains the same accuracy as the full SVM with merely 10% of the support vectors. Even when the cost budget is reduced to 1%, CVM can still yield a relatively high accuracy in all data sets.

4.6 Conclusion

We introduce CVM, a novel learning algorithm for reducing test-time classifier evaluation cost. Our algorithm focuses specifically on widely used kernel SVM classifiers and reduce the kernel SVM test-time evaluation cost. The algorithm builds upon regular SVMs, and compresses the model with evaluation cost constraint by solving the exact SVM optimization problem with a forward selection algorithm. We further optimize the compressed model by searching a better basis (support vectors) to approximate the decision boundaries formed by full SVM solution in a continuous input space. We demonstrate that only a small set of these optimized support vectors can generate very accurate predictions on a variety of data sets.

Chapter 5

Conclusion

In real-world machine learning applications, such as email-spam [128], adult content filtering [40], and web-search engines [142], data sets are usually very large and tasks are frequently repeated. In applications of such large scale, computation cost must be budgeted and accounted for. In this thesis, we systematically investigate the test-time cost and introduce learning under test-time resource constraints, a new branch of machine learning that focuses mainly on learning efficient, low test-time cost classifiers.

We propose three strategies that promise large gains in reducing test-time cost if classifiers are trained explicitly to stay within pre-defined test-time budgets. 1) Feature extraction cost reduction; 2) Classification with trees and cascade; 3) Model compression.

The first strategy gives rise to solutions of two learning under test-time resource constraints scenarios: low feature extraction cost classification and anytime classification. Correspondingly, we propose two algorithms *Greedy Miser* and *Anytime Feature Representation Learning (AFR)* that employ the common reducing feature extraction cost strategy. *Greedy Miser* focuses on incorporating feature extraction cost during training an additive classifier and *AFR* aims to build an anytime classifier by learning an anytime feature representation. The second strategy results from a different goal, budgeting amortized test-time cost. It inspires two novel algorithms *Cost-sensitive Tree of Classifiers (CSTC)* and its cascade variant *Cost-sensitive Cascade of Classifiers (CSCC)*. CSTC builds a tree of classifiers that partitions the input space and learns specialized classifiers for different inputs, and CSCC is a variant of CSTC for binary imbalanced data sets. Finally, based on the last strategy, model compression, we propose a new *Compressed Vector Machine (CVM)* algorithm, which aims exclusively to compress the model of kernel SVMs, and budget its evaluation cost. CVM

cherry-picks a small subset of support vectors learned by the full SVM solution, and optimizes this subset of support vectors to approximate the decision boundary formed by the full model.

Addressing learning under test-time resource constraints in a principled fashion has high impact potential in two ways: i) reducing the cost required for the average case frees up more resources for the rare difficult cases – thus improving accuracy; ii) decreasing computational demands of massive industrial computations can substantially reduce energy consumption and greenhouse emissions.

As future works, we would like to explore incorporating latest computing technologies (such as parallel computing [121] and graphics processing unit (GPU) [79, 80]) during training classifiers to better trade off test-time cost and accuracy.

References

[1] U. Alon, N. Barkai, D. A. Notterman, K. Gish, S. Ybarra, D. Mack, and A. J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proceedings of the National Academy of Sciences*, 96(12):6745–6750, 1999.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[3] M. Belkin, P. Niyogi, and V. Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *The Journal of Machine Learning Research*, 7:2399–2434, 2006.

[4] S. Bengio, J. Weston, and D. Grangier. Label embedding trees for large multi-class tasks. *NIPS*, 23:163–171, 2010.

[5] J. F. Bonnans and A. Shapiro. Optimization problems with perturbations: A guided tour. *SIAM review*, 40(2):228–264, 1998.

[6] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge Univ Pr, 2004.

[7] L. Breiman. *Classification and regression trees*. Chapman & Hall/CRC, 1984.

[8] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] A. Broder, E. Gabrilovich, V. Josifovski, G. Mavromatis, D. Metzler, and J. Wang. Exploiting site-level information to improve web search. In *CIKM '10: Proceedings of the 19th ACM Conference on Information and Knowledge Management*, 2010.

[10] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[11] C. Burges. Simplified support vector decision rules. In *ICML*, volume 96, pages 71–77, 1996.

[12] C. J. Burges and B. Schölkopf. Improving the accuracy and speed of support vector machines. In *Advances in Neural Information Processing Systems*, pages 375–381, 1997.

[13] R. Busa-Fekete, D. Benbouzid, B. Kégl, et al. Fast classification using sparse decision dags. In *ICML*, 2012.

[14] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *WSDM'3*, pages 411–420, 2010.

[15] X. Chai, L. Deng, Q. Yang, and C. X. Ling. Test-cost sensitive naive bayes classification. In *Data Mining, 2004. ICDM'04.*, pages 51–58. IEEE, 2004.

[16] C. Chang and C. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[17] O. Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178, 2007.

[18] O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. In *JMLR: Workshop and Conference Proceedings*, volume 14, pages 1–24, 2011.

[19] O. Chapelle and A. Zien. Semi-supervised classification by low density separation. 2004.

[20] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1):131–159, 2002.

[21] O. Chapelle, B. Schölkopf, A. Zien, et al. *Semi-supervised learning*, volume 2. MIT press Cambridge, 2006.

[22] O. Chapelle, P. Shivaswamy, S. Vadrevu, K. Weinberger, Y. Zhang, and B. Tseng. Boosted multi-task learning. *Machine Learning*, pages 1–25, 2010.

[23] O. Chapelle, P. Shivaswamy, S. Vadrevu, K. Weinberger, Y. Zhang, and B. Tseng. Boosted multi-task learning. *Machine learning*, 85(1):149–173, 2011.

[24] M. Chen, Z. Xu, K. Q. Weinberger, and O. Chapelle. Classifier cascade for minimizing feature evaluation cost. In *AISTATS*, 2012.

[25] W. Chen, Y. Chen, K. Q. Weinberger, Q. Lu, and X. Chen. Maximum variance correction with application to a* search. In *Proc. of 27th AAAI Conference on Artificial Intelligence (AAAI)*, 2013.

[26] W. Chen, K. Q. Weinberger, and Y. Chen. Maximum variance correction with application to a* search. In *Proc. of 30th Intl. Conf. on Machine Learning (ICML)*, 2013.

[27] W. Chen, Y. Chen, and K. Q. Weinberger. Fast flux discriminant for large-scale sparse nonlinear classification. In *Proc. of 20th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD)*, 2014.

[28] F. Chung. *Spectral graph theory*, volume 92. AMS Bookstore, 1997.

[29] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[30] R. W. Cottle. Manifestations of the schur complement. *Linear Algebra and its Applications*, 8(3):189–211, 1974.

[31] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[32] O. Dekel, S. Shalev-Shwartz, and Y. Singer. The forgetron: A kernel-based perceptron on a budget. *SIAM Journal on Computing*, 37(5):1342–1372, 2008.

[33] J. Deng, S. Satheesh, A. C. Berg, and L. Fei-Fei. Fast and balanced: Efficient label tree learning for large scale object recognition. In *NIPS*, 2011.

[34] M. Dredze, R. Gevaryahu, and A. Elias-Bachrach. Learning fast classifiers for image spam. In *proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2007.

[35] H. Drucker, D. Wu, and V. N. Vapnik. Support vector machines for spam categorization. *Neural Networks, IEEE Transactions on*, 10(5):1048–1054, 1999.

[36] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the l 1-ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM, 2008.

[37] M. M. Dundar and J. Bi. Joint optimization of cascaded classifiers for computer aided detection. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[38] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[39] J. A. Etzel, V. Gazzola, and C. Keysers. An introduction to anatomical roi-based fmri classification analysis. *Brain Research*, 1282:114–125, 2009.

[40] M. Fleck, D. Forsyth, and C. Bregler. Finding naked people. *ECCV*, pages 593–602, 1996.

[41] Y. Freund and R. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[42] Y. Freund, R. Schapire, and N. Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[43] J. Friedman, T. Hastie, and R. Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

[44] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *The Annals of Statistics*, pages 1189–1232, 2001.

[45] T. Gao and D. Koller. Active classification based on value of classifier. In *NIPS*, pages 1062–1070. 2011.

[46] T. Gao and D. Koller. Multiclass boosting with hinge loss based on output coding. ICML '11, pages 569–576, 2011.

[47] D. Gavrila. Pedestrian detection from a moving vehicle. *ECCV 2000*, pages 37–49, 2000.

[48] G. H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins University Press, 1996.

[49] A. Grubb and J. A. Bagnell. Generalized boosting algorithms for convex optimization. *arXiv preprint arXiv:1105.2054*, 2011.

[50] A. Grubb and J. A. Bagnell. Speedboost: Anytime prediction with uniform near-optimality. In *AISTATS*, 2012.

[51] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[52] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[53] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning*. Springer, 2009.

[54] B. Heisele, P. Ho, and T. Poggio. Face recognition with support vector machines: Global versus component-based approach. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 688–694. IEEE, 2001.

[55] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[56] D. W. Hosmer and S. Lemeshow. *Applied logistic regression*, volume 354. Wiley-Interscience, 2000.

[57] J. Huang, T. Zhang, and D. Metaxas. Learning with structured sparsity. *The Journal of Machine Learning Research*, 12:3371–3412, 2011.

[58] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM TOIS*, 20(4):422–446, 2002.

[59] T. Joachims. Transductive inference for text classification using support vector machines. In *ICML*, volume 99, pages 200–209, 1999.

[60] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[61] S. Karayev, T. Baumgartner, M. Fritz, and T. Darrell. Timely object recognition. In *Advances in Neural Information Processing Systems 25*, pages 899–907, 2012.

[62] D. Kedem, S. Tyree, K. Q. Weinbeger, F. Sha, and G. Lanckriet. Non-linear metric learning. In *NIPS*, pages 2582–2590. 2012.

[63] S. Keerthi. Generalized lars as an effective feature selection tool for text classification with svms. In *Proceedings of the 22nd international conference on Machine learning*, pages 417–424. ACM, 2005.

[64] S. Keerthi, O. Chapelle, and D. DeCoste. Building support vector machines with reduced classifier complexity. *The Journal of Machine Learning Research*, 7:1493–1515, 2006.

[65] G. S. Kimeldorf and G. Wahba. A correspondence between bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, pages 495–502, 1970.

[66] D. Kleinbaum, L. Kupper, A. Nizam, and E. Rosenberg. *Applied regression analysis and other multivariable methods*. Cengage Learning, 2013.

[67] J. K. Kolter and A. Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 521–528. ACM, 2009.

[68] B. Korte and J. Vygen. *Combinatorial optimization*, volume 1. Springer, 2002.

[69] M. Kowalski. Sparse regression using mixed norms. *Applied and Computational Harmonic Analysis*, 27(3):303–324, 2009.

[70] G. Lanckriet, L. Ghaoui, C. Bhattacharyya, and M. I. Jordan. A robust minimax approach to classification. *The Journal of Machine Learning Research*, 3:555–582, 2003.

[71] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, pages 2169–2178, 2006.

[72] L. Le Cam. Asymptotic methods in statistical decision theory. *New York*, 1986.

[73] S. Lee, H. Lee, P. Abbeel, and A. Y. Ng. Efficient l˜ 1 regularized logistic regression. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 401. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[74] L. Lefakis and F. Fleuret. Joint cascade optimization using a product of boosted classifiers. In *NIPS*, pages 1315–1323. 2010.

[75] E. L. Lehmann and G. Casella. *Theory of point estimation*, volume 31. Springer, 1998.

[76] L. J. Li, H. Su, E. P. Xing, and L. Fei-Fei. Object bank: A high-level image representation for scene classification and semantic feature sparsification. *NIPS*, 2010.

[77] Y. Liu, M. Sharma, C. Gaona, J. Breshears, J. Roland, Z. Freudenburg, E. Leuthardt, and K. Q. Weinberger. Decoding ipsilateral finger movements from ecog signals in humans. In *Advances in Neural Information Processing Systems*, pages 1468–1476, 2010.

[78] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[79] L. Ma, K. Agrawal, and R. D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, January 2014.

[80] L. Ma, K. Agrawal, and R. D. Chamberlain. Analysis of classic algorithms on GPUs. In *Proc. of the 12th ACM/IEEE Int'l Conf. on High Performance Computing and Simulation (HPCS)*, 2014.

[81] S. Ma, X. Song, and J. Huang. Supervised group lasso with applications to microarray data analysis. *BMC bioinformatics*, 8(1):60, 2007.

[82] P. Melville, N. Shah, L. Mihalkova, and R. J. Mooney. Experiments on ensembles with missing and noisy data. In *Multiple Classifier Systems*, pages 293–302. Springer, 2004.

[83] A. Mohan, Z. Chen, and K. Q. Weinberger. Web-search ranking with initialized gradient boosted regression trees. *JMLR: Workshop and Conference Proceedings*, 14:77–89, 2011.

[84] E. A. Nadaraya. On estimating regression. *Theory of Probability & Its Applications*, 9 (1):141–142, 1964.

[85] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[86] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell. Text classification from labeled and unlabeled documents using em. *Machine learning*, 39(2-3):103–134, 2000.

[87] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

[88] F. Pan, T. Converse, D. Ahn, F. Salvetti, and G. Donato. Feature selection for ranking using boosted trees. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2025–2028. ACM, 2009.

[89] M. Y. Park and T. Hastie. L1-regularization path algorithm for generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(4):659–677, 2007.

[90] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(8):1226–1238, 2005.

[91] S. Perkins, K. Lacker, and J. Theiler. Grafting: Fast, incremental feature selection by gradient descent in function space. *The Journal of Machine Learning Research*, 3: 1333–1356, 2003.

[92] K. B. Petersen and M. S. Pedersen. The matrix cookbook, oct 2008.

[93] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. 1999.

[94] J. C. Platt, N. Cristianini, and J. Shawe-taylor. Large margin dags for multiclass classification. In *Advances in Neural Information Processing Systems 12*, 2000.

[95] E. Polak. *Computational methods in optimization: a unified approach*, volume 77. Academic press, 1971.

[96] J. Pujara, H. Daumé III, and L. Getoor. Using classifier cascades for scalable e-mail classification. In *CEAS*, 2011.

[97] A. Rakotomamonjy, F. Bach, S. Canu, Y. Grandvalet, et al. Simplemkl. *Journal of Machine Learning Research*, 9:2491–2521, 2008.

[98] C. E. Rasmussen. Gaussian processes for machine learning. 2006.

[99] V. C. Raykar, B. Krishnapuram, and S. Yu. Designing efficient cascaded classifiers: tradeoff between accuracy and cost. In *ACM SIGKDD*, pages 853–860, 2010.

[100] L. Reyzin. Boosting on a budget: Sampling for feature-efficient prediction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 529–536, 2011.

[101] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *The Journal of Machine Learning Research*, 5:101–141, 2004.

[102] S. Rosset, J. Zhu, and T. Hastie. Boosting as a regularized path to a maximum margin classifier. *The Journal of Machine Learning Research*, 5:941–973, 2004.

[103] V. Roth. The generalized lasso. *Neural Networks, IEEE Transactions on*, 15(1):16–28, 2004.

[104] V. Roth and B. Fischer. The group-lasso for generalized linear models: uniqueness of solutions and efficient algorithms. In *Proceedings of the 25th international conference on Machine learning*, pages 848–855. ACM, 2008.

[105] M. Saberian and N. Vasconcelos. Boosting classifier cascades. In *NIPS*, pages 2047–2055. 2010.

[106] Y. Saeys, I. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[107] R. Salakhutdinov and G. E. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *International Conference on Artificial Intelligence and Statistics*, pages 412–419, 2007.

[108] R. E. Schapire. A brief introduction to boosting. In *IJCAI*, volume 16, pages 1401–1406, 1999.

[109] B. Schölkopf. The kernel trick for distances. *Advances in neural information processing systems*, pages 301–307, 2001.

[110] B. Schölkopf and A.J. Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[111] C. Shawe-Taylor, B. Schölkopf, and A. Smola. The support vector machine. 2000.

[112] V. Sindhwani and S. Keerthi. Large scale semi-supervised linear svms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 477–484. ACM, 2006.

[113] V. Sindhwani, P. Niyogi, and M. Belkin. Beyond the point cloud: from transductive to semi-supervised learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 824–831. ACM, 2005.

[114] L. Song, A. Smola, A. Gretton, J. Bedo, and K. Borgwardt. Feature selection via dependence maximization. *The Journal of Machine Learning Research*, 98888:1393–1434, 2012.

[115] S. Sra. Fast projections onto 1, q-norm balls for grouped feature selection. In *Machine Learning and Knowledge Discovery in Databases*, pages 305–317. Springer, 2011.

[116] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[117] K. Trapeznikov, V. Saligrama, and D. Castañón. Multi-stage classifier design. *Machine learning*, 92(2-3):479–502, 2013.

[118] T. Trzcinski, M. Christoudias, V. Lepetit, and P. Fua. Learning image descriptors with the boosting-trick. In *NIPS*, pages 278–286. 2012.

[119] E. Tuv, A. Borisov, G. Runger, and K. Torkkola. Feature selection with ensembles, artificial variables, and redundancy elimination. *The Journal of Machine Learning Research*, 10:1341–1366, 2009.

[120] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *WWW*, pages 387–396. ACM, 2011.

[121] S. Tyree, J. R. Gardner, K. Q. Weinberger, K. Agrawal, and J. Tran. Parallel support vector machines in practice. *arXiv preprint arXiv:1404.1066*, 2014.

[122] V. Vapnik. *Statistical Learning Theory*. Wiley, N.Y., 1998.

[123] V. Vapnik. *The nature of statistical learning theory*. springer, 2000.

[124] A. Vedaldi and B. Fulkerson. Vlfeat: An open and portable library of computer vision algorithms. In *Proceedings of the international conference on Multimedia*, pages 1469–1472. ACM, 2010.

[125] P. Viola and M.J. Jones. Robust real-time face detection. *IJCV*, 57(2):137–154, 2004.

[126] J. Wang and V. Saligrama. Local supervised learning through space partitioning. In *NIPS*, pages 91–99, 2012.

[127] Z. Wang, K. Crammer, and S. Vucetic. Breaking the curse of kernelization: Budgeted stochastic gradient descent for large-scale svm training. *Journal of Machine Learning Research*, 13:3103–3131, 2012.

[128] K. Q. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *ICML*, pages 1113–1120, 2009.

[129] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggio, and V. Vapnik. Feature selection for svms. In *NIPS*, volume 12, pages 668–674, 2000.

[130] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *CVPR*, pages 3485–3492. IEEE, 2010.

[131] Z. Xu, I. King, T. Lyu, and R. Jin. Discriminative semi-supervised feature selection via manifold regularization. *Neural Networks, IEEE Transactions on*, 21(7):1033–1047, 2010.

[132] Z. Xu, K. Q. Weinberger, and O. Chapelle. The greedy miser: Learning under test-time budgets. In *ICML*, pages 1175–1182, 2012.

[133] Z. Xu, K. Q. Weinberger, and O. Chapelle. Distance metric learning for kernel machines. *arXiv preprint arXiv:1208.3422*, 2012.

[134] Z. Xu, M. J. Kusner, M. Chen, and K. Q. Weinberger. Cost-sensitive tree of classifiers. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 133–141. JMLR Workshop and Conference Proceedings, 2013.

[135] Z. Xu, M. J. Kusner, G. Huang, and K. Q. Weinberger. Anytime representation learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1076–1084, 2013.

[136] Z. Xu, G. Huang, K. Q. Weinberger, and A. X. Zheng. Gradient boosted feature selection. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 522–531. ACM, 2014.

[137] Z. Xu, M. J. Kusner, K. Q. Weinberger, M. Chen, and O. Chapelle. Classifier cascades and trees for minimizing feature evaluation cost. *Journal of Machine Learning Research*, 15:2113–2144, 2014.

[138] M. Yamada, W. Jitkrittum, L. Sigal, E. P. Xing, and M. Sugiyama. High-dimensional feature selection by feature-wise non-linear lasso. *arXiv preprint arXiv:1202.0515*, 2012.

[139] M. Yang. Mixtures of linear subspaces for face detection. 1999.

[140] K. Yoshiyama and A. Sakurai. Manifold-regularized minimax probability machine. In *Partially Supervised Learning*, pages 42–51. Springer, 2012.

[141] T. Zhang. Multi-stage convex relaxation for learning with sparse regularization. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1929–1936. 2008.

[142] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. In *NIPS*, pages 1697–1704. Cambridge, MA, 2008.

[143] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.